# REAL-TIME GPS-ALTERNATIVE NAVIGATION USING COMMODITY HARDWARE

THESIS

Jordan L. Fletcher, Captain, USAF

AFIT/GCS/ENG/07-02

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

**AIR FORCE INSTITUTE OF TECHNOLOGY**

**Wright-Patterson Air Force Base, Ohio**

AFIT/GCS/ENG/07-02

# REAL-TIME GPS-ALTERNATIVE NAVIGATION USING COMMODITY HARDWARE

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Jordan L. Fletcher, BS

Captain, USAF

June 2007

AFIT/GCS/ENG/07-02

# REAL-TIME GPS-ALTERNATIVE NAVIGATION USING COMMODITY HARDWARE

Jordan L. Fletcher, BS

Captain, USAF

Approved:

| | |
|---|---|
| //signed// | |
| Michael Veth, Maj, USAF (Chairman) | 4 June 2007 |
| //signed// | |
| John Raquet, Ph D. (Member) | 4 June 2007 |
| //signed// | |
| Gilbert L. Peterson, Ph D. (Member) | 4 June 2007 |
| //signed// | |
| Guna Seetharaman, Ph.D. (Member) | 4 June 2007 |

## Acknowledgments

I would like to express my sincere appreciation to my thesis advisors, Maj Michael Veth and Dr. John Raquet for their guidance and support throughout the course of this thesis effort. Your knowledge of navigation is truly awe-inspiring. I only wish there were 48 hours in a day, or two of me, so I could work on everything we wanted to work on.

I would like to thank my girlfriend. I'm sorry about the dishes and the chores for the past year. I promise to make it up to you in California. I would also like to thank Don, John and Jared from the ANT Center for their help with hardware, MATLAB and their impressive GPS knowledge. I also appreciate you serving as my personal sounding board for the past year. Good luck with the UGC.

Lastly, I would like to thank my sponsors at the Air Force Research Labs Munitions Directorate for their support and funding of my research activities.

Jordan L. Fletcher

## Table of Contents

## List of Figures

## List of Tables

AFIT/GCS/ENG/07-02

## Abstract

Modern navigation systems can use the Global Positioning System (GPS) to accurately determine position with precision in some cases bordering on millimeters. Unfortunately, GPS technology is susceptible to jamming, interception, and unavailability indoors or underground. There are several navigation techniques that can be used to navigate during times of GPS unavailability, but there are very few that result in GPS-level precision. One method of achieving high precision navigation without GPS is to fuse data obtained from multiple sensors.

This thesis explores the fusion of imaging and inertial sensors and implements them in a real-time system that mimics human navigation. In addition, programmable graphics processing unit technology is leveraged to perform stream-based image processing using a computer's video card. The resulting system can perform complex mathematical computations in a fraction of the time those same operations would take on a CPU-based platform. The resulting system is an adaptable, portable, inexpensive and self-contained software and hardware platform, which paves the way for advances in autonomous navigation, mobile cartography, and artificial intelligence.

# REAL-TIME GPS-ALTERNATIVE NAVIGATION USING COMMODITY HARDWARE

# I. Introduction

## 1.1 The Need for Precision Navigation

Korean Air Lines flight KAL 007 took off from John F. Kennedy International Airport, the morning of August 31, 1983, destined for Seoul-Kimpo International Airport. By 18:26 GMT, KAL 007 was unknowingly off-course, 500 kilometers deep in Soviet Territory and flanked by two Soviet Su-15 jet fighter planes. Minutes later, KAL 007 and its 269 passengers and crew crashed into the sea north of Moneron Island, shot down by a missile fired from a Soviet fighter.

This tragic incident could have been avoided using advanced navigation techniques available today. The advent of the Global Positioning System (GPS) has made precision navigation more affordable, portable, and commonplace on ships, airplanes, and automobiles (Kaplan 2005). However, GPS can have significant problems with availability and accuracy. Specifically, current GPS receivers cannot receive signals indoors, underground, or on other planets. In addition, problems with jamming (Boyle 2003), multipath (Georgiadou 1988), atmospheric dispersion (Snay 2000), and solar flares (Chen 2005) can have potentially dangerous effects on the accuracy of a position obtained from GPS. These problems have led to a growing demand for GPS-alternative precision navigation techniques in both the civilian and military community.

Precision navigation also paves the way for unmanned vehicles, which is a topic of interest for the United States military. In the National Defense Authorization Act for Fiscal Year 2001, Congress mandated that "It shall be a goal of the Armed Forces to achieve the fielding of unmanned, remotely controlled technology such that… by 2015, one-third of the operational ground combat vehicles are unmanned." Agencies like the Defense Advanced Research Projects Agency (DARPA) conduct annual precision navigation competitions in support of this Congressional mandate. Every "dull, dirty, or dangerous" task that can be carried out using a machine instead of a human protects our warfighters and allows valuable human resources to be used more effectively (DARPA 2007). However, any precision navigation solution will have to be robust to the military environment and cost-effective for ground vehicle applications.

## 1.2 Research Goals and Hypothesis

The thrust of this research was to improve a state-of-the-art image-aided inertial navigation system developed in previous research at AFIT (Veth 2006). Using image sensors for precision navigation has outstanding potential. However, previous applications have been limited due to the complex image processing required and the corresponding computational requirements. While the previous system overcame several of the classic image processing barriers and calculated position with near-GPS level accuracy, it was incapable of real-time processing using mobile hardware. Therefore, the goal of this research was to approach the problem from a software engineering standpoint and resolve the real-time processing and mobility shortcomings of the previous navigation system. To achieve this goal, the fundamental idea was to exploit the parallel nature of both the multiple sensor data acquisition and the image processing algorithm used by the navigation system. The hypothesis was that the computational burden of the image processing algorithm could be offloaded to a graphics processing unit, and multi-sensor data acquisition and access could be optimized using proven software engineering techniques.

## 1.3 Scope of Research

The navigation research conducted in this thesis was performed as a continuation and improvement to doctoral research previously conducted at AFIT (Veth 2006). The previous research in the areas of image-aided inertial navigation was novel, and even now is state-of-the-art GPS-alternative navigation. The previous system was built to test and demonstrate image-aided inertial navigation theory, with a focus on completeness and accuracy. The work presented in this thesis is aimed at continuing the development of the previous system, but focuses on cost and performance. In addition, this thesis is aimed at transitioning the previous work to a solution that will perform navigation predictions as accurately as the previous system, but in real-time on commodity computing hardware.

Likewise, the general-purpose graphics processing unit (GPGPU) concepts presented in this thesis are built upon a larger, existing knowledge base. Most of the GPGPU concepts were derived from the GPGPU tutorial and other works from (GPGPU 2007). The GPGPU software package used in this thesis was built upon OpenVIDIA, an image processing application written by James Fung (Fung 2005). Fung's software was researched, analyzed, modified for navigation use, and finally integrated into the software system architecture.

Furthermore, the multi-sensor fusion algorithm used in this thesis is also not a novel concept. Multi-sensor fusion was used as the foundation to optimally combine imaging and inertial sensors on a single platform. This feat was accomplished by fusing the blackboard architecture described in (Dong 2005) with the Model-View-Controller architecture described in (Gamma 1995).

The navigation system presented in this thesis is a synergistic combination of many different theories and concepts that have never before been applied to a navigation problem. However, this thesis does not cover any of the afore-mentioned concepts in minute detail. Therefore, readers who are looking for in-depth information on precision navigation, GPGPU, or multi-sensor fusion should read the works by the referenced authors. Readers who want to know how these concepts can be combined in a real-time system for navigation should read on.

## 1.4 Related Research

Precision navigation has been a persistent topic of interest in the defense and civilian industry. GPS has proved invaluable in aircraft, automobile and ship navigation, and is finding more use in manufacturing, surveying and agriculture (Shanwad 2002). Naturally, most current precision navigation research has been improvements in GPS accuracy and availability. However, there are also a few researchers and scientists who have been developing GPS-alternative navigation techniques and platforms. These scientists are mostly interested in solving the Simultaneous Localization and Mapping

Problem, or SLAM. Current research in SLAM, as well as a conceptual overview of the problem, has been compiled in a paper by Chen (Chen 2007). The navigation system presented in this thesis can also be applied to the SLAM problem, but that is not the focus of this research.

First and foremost, the navigation system presented in this paper is based upon previous research at AFIT. The navigation system is probability-based, using a Kalman Filter to combine image and inertial sensor readings into a single navigation prediction. This work is significant in the fact that it is one of the first systems to use the Scale-Invariant Feature Transform (SIFT) algorithm (Lowe 2004) to choose landmarks for navigation in an unknown environment, described in Chapter 2. SIFT landmarks are unique from each other, so they can potentially be saved in a database for future navigation or map-building. Also significant in the previous work is the way in which a 3D representation of the navigating body and landmarks are constructed from 2D imagery. This approach is more sophisticated than the work of many others in this area, because it can be applied to airborne vehicles in addition to ground-based navigating bodies. However, the work is limited to strictly post-processing navigation data, since it does not have the ability to simultaneously acquire sensor data and process navigation predictions.

Sebastian Thrun, of Stanford University, has developed many vision-based navigation algorithms and platforms in the past half-decade, including MINERVA (Thrun 2000) and STANLEY (Thrun 2006). Much of his research is also probability-based using a Kalman filter, similar to the navigation system in this paper and the work at

AFIT. However, none of his systems take advantage of the parallel nature of multi-sensor fusion or image processing. Instead, his approaches use a simplified form of image processing and feature extraction to obtain environmental landmarks. Although Thrun’s system can perform basic navigation tasks like obstacle avoidance, his systems do not have the capability to navigate precisely in unknown environments.

Stefano Panzieri has also done significant research in the area of controls, robotics, and sensor fusion. His SLAM work has ranged from particle filter-based implementations (Panzieri 2006) to Kalman filter-based implementations (Panzieri 2003) and many methods in-between. Panzieri has also performed a significant amount of his research in the field of vision-based navigation using low-cost sensors (Panzieri 2005), which is similar to a secondary focus of this thesis. However, Panzieri does not make use of the SIFT algorithm or the parallel nature of sensor acquisition and image processing, as the navigation system presented in this paper does. His focus is instead on using landmarks in known environments, such as rectangular light fixtures in an indoor environment, to ease the computational burden and complexity of image processing.

Alberto Broggi is primarily concerned with the use of vision for vehicle applications. His research is significant in that his sensor platforms fuse primary vision sensors with laser (Broggi 2006) and radar (Broggi 2006b) sensors to obtain terrain and range measurements for navigation. This approach is very effective, but once again does not exploit parallelism and is not designed for concurrent data acquisition. Although Broggi’s approaches construct a 3D representation of the 2D stereo image sensor data, just as the approach in this thesis does, the 3D objects in Broggi’s system are merely used for obstacle avoidance by the navigating platform, rather than as unique landmarks, which is useful to problems like SLAM and target recognition.

Other notable names in the GPS-alternative navigation field include Robert Sim and Massimo Bertozzi, who used methods similar to those described above for SLAM solutions. As evidenced above, there are no researchers who are working on real-time precision image-aided navigation in the manner described in this thesis. Researchers who are performing work in this area rely upon less complex features or *a-priori* feature and environment information to perform image processing in real-time. This makes the work in this thesis valuable to the navigation community and truly state-of-the-art.

There have been a few attempts to augment CPU-based image processing by using programmable hardware. The objective of this type of research is to speed up feature extraction by using hardware dedicated to image processing. Stephen Se from the University of British Columbia has developed a field-programmable gate array (FPGA) implementation of SIFT (Se 2004). The advantage of the FPGA is that it can perform very fast computations via the use of dedicated, specialized hardware tailored for an arbitrary application. Se’s work resulted in a system that was able to create a 3D terrain model using stereo images captured at 7 Hertz (Hz) and 500 x 450 resolution using an FPGA and other dedicated hardware. This was a speedup of 3.5 times the rate at which his CPU-only solution could perform the task. However, use of specialized hardware, such as an integrated circuit, imposes a constraint on the flexibility and extensibility of the system. In addition, the cost for programmable FPGA hardware is prohibitive to proliferating these devices on a ground vehicle fleet.

There has also been a tremendous amount of recent work in the field of GPGPU image processing research. The most notable research for this thesis is the work done by James Fung (Fung 2005), Sudipta Sinha (Sinha 2006) and Sebastian Heymann (Heymann 2007). These three individuals, all working on separate projects at different universities and industries, have independently developed GPU-based SIFT implementations. All have found fantastic success with their implementations, with an increase in feature extraction speed between 4 and 14 times the speed of CPU feature extraction. In addition, Fung's work is open-source, meaning it is available and free for public use. The price/performance ratio is also more favorable in the case of GPU-based SIFT vs. FPGA-based SIFT, making it an interesting research area to explore. While their work is valuable to the image processing community, their implementations require adaptation for use in precision navigation applications.

## 1.5 Thesis Overview

The remainder of this thesis is divided into four Chapters. Chapter 2 provides background information on technologies used in the navigation system. This includes background information on the Scale-Invariant Feature Transform (SIFT) used to find landmarks/features from captured images and general-purpose graphics processing unit (GPGPU) technology used for hardware-accelerated image processing. Chapter 2 also provides background information on real-time image processing, multi-sensor fusion theory, and a background to software engineering design principles, all of which are fundamental to this problem. Chapter 3 describes how the software package was created, how hardware was chosen and integrated, and how experiments were designed. Chapter

4 contains the results of the experiments, including the speedup of GPGPU accelerated feature extraction over CPU feature extraction, and offers explanations for the results. Chapter five summarizes the process, set-up and results, and details future research that can be done in this area.

# II. Literature Review

This chapter provides background information for precision navigation techniques at a level necessary to understand the navigation system presented in this thesis. This is followed by an introduction to biologically-inspired navigation systems considered for GPS-alternative solutions. Next, the graphics processing unit (GPU) is described in detail, including components, stream processing model, rendering pipeline and the concept of general purpose computation using GPU hardware. This is followed by a review of real-time systems and real-time constraints. The final section of this chapter is a review of software engineering principles that are vital to understanding the software architecture employed for this problem.

## 2.1 Introduction to Precision Navigation

Precision navigation is a term used to describe navigation that requires a high level of accuracy. The required accuracy can range from meters to millimeters, depending on the application. For instance, the precision needed for row-spacing in agriculture can be on the order of meters. If a seed is planted a meter away from where it was supposed to be planted, there may be crop loss, but there will likely be no loss of life. However, precision approach and landing systems and munitions guidance systems require sub-meter accuracy. An aircraft that lands a meter away from where it was supposed to land could very likely result in loss of life. There are many ways to perform precision navigation, including three mentioned in this thesis: inertial navigation, image-aided navigation, and the global positioning system.

### 2.1.1 Inertial Navigation Systems

Inertial navigation systems (INS) vary in size, architecture, and precision, but all use an inertial measurement unit (IMU) for sensor readings. The most basic IMUs contain a clock for timing, accelerometers to determine linear acceleration and gyroscopes to determine the angular rotation rate of the INS relative to some inertial reference frame. All INS are based on the principle of dead-reckoning. Dead reckoning refers to the use of past measurements of elapsed time, speed and heading to predict one's current position. As illustrated in Figure 2.1, an INS can determine position, velocity, orientation and angular velocity using only the IMU readings and elapsed time.

The inherent problem with dead reckoning is that over time, inaccurate or imprecise measurements from the IMU are integrated into larger errors in velocity and orientation and even larger errors in position and attitude estimates (Judd 1997). Typically, the precision and accuracy of an INS decrease as the size of an INS decreases, making these devices a trade-off of mobility for navigation performance. Sometimes, the INS sensors are inaccurate and have false readings. Other times, their readings simply aren't precise enough to capture minute changes in pose or position of the navigation body. Either way, these problems are compounded during dead-reckoning and can result in poor navigation performance.

INS inaccuracy can be compensated for using a few standard approaches. One standard approach is to model the inaccuracy of the sensor, and adjust the sensor readings accordingly to compensate for the inaccuracy. The inaccuracy of the sensor can come from many sources, as described in (Veth 2006:19). Each source of error has to be determined and compensated for in the IMU model. However, an IMU cannot determine its error on its own.

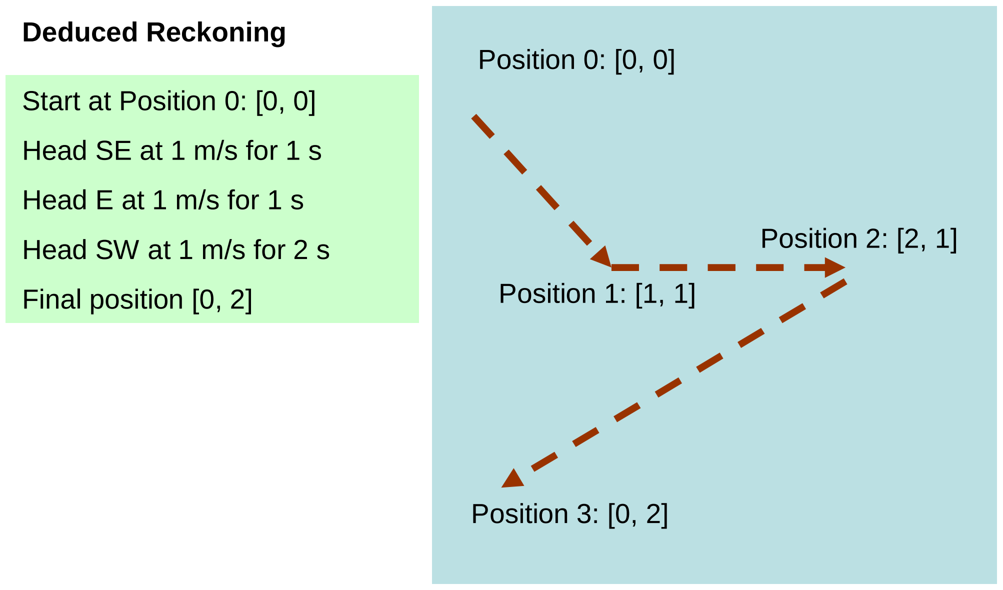


Figure 2.1: Simple INS example using dead reckoning. By integrating the sensor readings (left) over time, the final position can be computed. The path is re-created on the right for a visual map of the route taken.

One of the most common ways to determine IMU sensor error and compensate for it in a ground-based platform is by using a zero-velocity update. The principle behind the zero-velocity update is to stop all motion of the navigating platform, and subtract any acceleration or velocity readings from the IMU. If the velocity and acceleration readings are constant, than the IMU likely has a constant source of error, or bias, that can be removed from future readings as well.

Another way to determine IMU error and compensate for inaccuracy is to use additional sensors. The readings from additional sensors can be combined with the IMU readings to come up with a unified navigation prediction. There are many ways to combine data from multiple sensors. One approach that has been applied in many systems is to use a Kalman filter.

### 2.1.2 Kalman Filter and Multi-Sensor Fusion

The Kalman filter is a recursive filter used to estimate the state of a dynamic system from incomplete, noisy measurements. In addition, the Kalman filter computes the estimated uncertainty of the state estimate using the state covariance matrix. The function of the Kalman filter can be divided into two distinct operations; propagation and measurement updates. Assuming that measurements are available at discrete time increments, ($t_i$, $t_{i+1}$, $t_{i+2}$ … $t_{i+n}$), the current state $Xt_i$ and state covariance, $Pt_i$ can be estimated using the state immediately prior to $t$, $Xt_{i-1}$, input measurements and noise. The second phase of a Kalman filter is the propagation phase, which uses the solutions from the measurement update phase above to propagate to the next update time, $Xt_{i+1}$. The Kalman filter equations and further information is available in (Maybeck 1979).

The Kalman filter can be used to incorporate readings from several measurements into a single, unified state estimate. In this way, the Kalman filter provides a mechanism for multi-sensor fusion. The measurement update phase can be performed for each measurement available at any discrete time interval. The propagation phase can then be invoked to determine the fused state estimate. In addition, the covariance matrix can be applied to each measurement source to determine the uncertainty associated with each sensor. For instance, if an inaccurate sensor, such as a consumer-grade IMU, was used as one measurement input, and a more accurate, tactical-grade IMU was used as a second input, then the uncertainty associated with the consumer-grade IMU would be greater than the uncertainty associated with the tactical-grade IMU. The readings from each IMU could then be weighted by the uncertainty for each sensor, resulting in a state estimate that was a weighted mean between the two sensors. The multi-sensor Kalman filter is shown in Figure 2 and described in more detail in (Veth 2006).

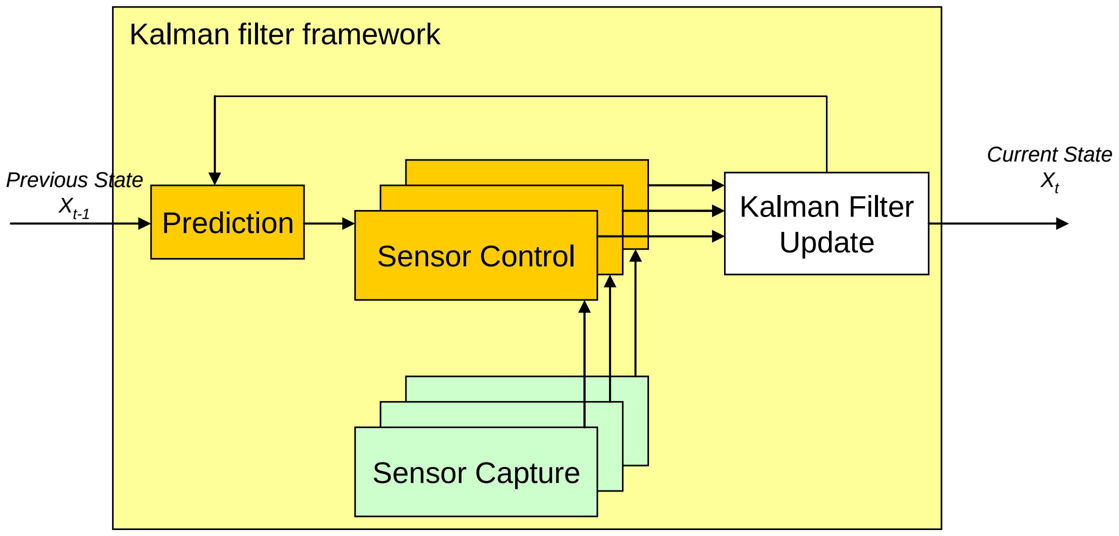


Figure 2.2: Multi-sensor Kalman Filter. The current state estimate, $X_t$ and the covariance error matrix, $P_t$, are computed using the previous state, $X_{t-1}$ and measurements from multiple sensors.

A multi-sensor fusion system is not limited to fusing sensor data from the same type of sensor. For example, a GPS navigation system can provide an INS with a position update at fixed intervals. By properly combining the GPS and INS data, the IMU errors in position and velocity can be stabilized. Examples of GPS-aided INS systems can be found in (Panzieri 2002) and (Pinto 2002). Image sensors can also be used to improve dead-reckoning navigation predictions when using an INS, as demonstrated in vision-based inertial systems by Thrun (Thrun 2000) and Sim (Sim 2005b). However, to understand how vision sensors can be used to correct an INS, one must first understand how vision sensors process information and use it for navigation.

### 2.1.3 Image-aided Navigation Systems and SIFT

Image sensors are found on many computing and navigation platforms (Bellini 2002; Burschka 2004; Mourikis 2007). To use these sensors effectively, one must understand some image processing and machine vision basics. The following section describes image processing basics, followed by image processing techniques for navigation, including feature extraction. This section describes why features are useful for landmark-based navigation, and how to use landmarks for image correspondence. Finally, the feature extraction algorithm used in this navigation system is described and evaluated for its applicability to navigation.

The concept of an image, as used in the context of this thesis, is a distribution of intensity values projected on a two-dimensional (2-D) plane. This 2-D plane can be a sheet of paper, a computer screen, or even a face of a cube. Typically, the intensity values are converted to a numeric representation whose values are in the range of the precision needed for the image processing application. One constrains the range of values by limiting the number of bits per intensity value, thus limiting the numeric precision used to describe each pixel. For example, if the image processing algorithm was for text recognition, an acceptable range for intensity values could be (0, 1), to indicate the presence or absence of intensity (or color) at each pixel. Thus one could say that a text recognition algorithm only needs 1-bit precision, because the intensity values could be constrained to 1 bit per pixel (bpp). Precision is tied to the amount of data that can be represented per pixel. A more precise image that had 4-bit precision could take on values from 0 to 16, which is equivalent to 16 possible intensity values (or 16 colors) per pixel. To illustrate this effect, compare an image with 1-bit per pixel (bpp) precision (2 possible values per pixel), to the identical image with 8-bit precision (256 possible values per pixel) in Figure 2.3.

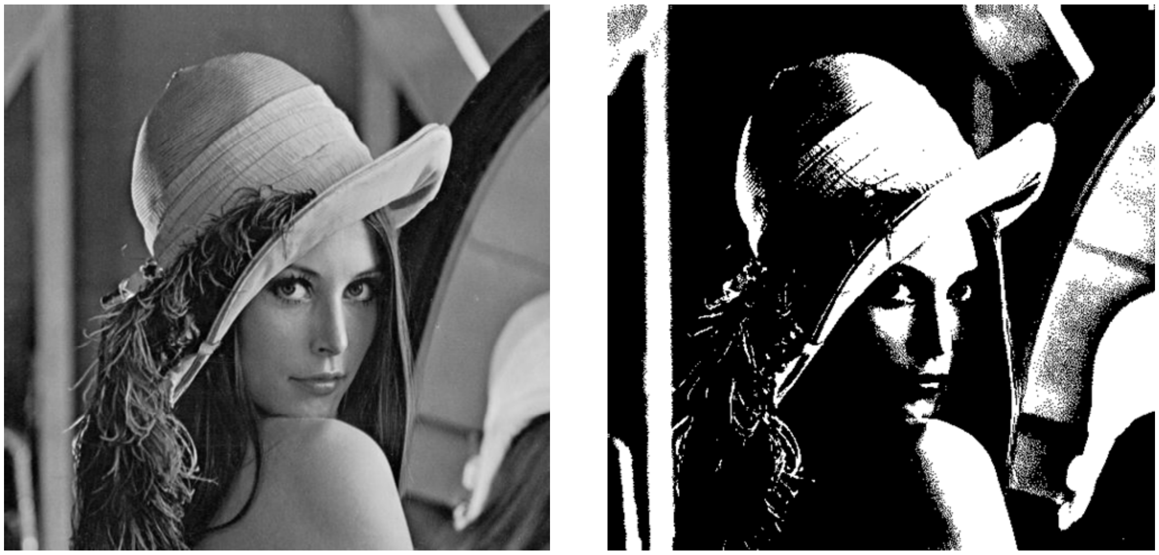


Figure 2.3: Image precision comparisons. The left image is 8 bpp and the right image is 1 bpp precision. More information can be obtained from images with greater precision.

In more advanced image systems, such as the human eye, the intensity value at any pixel can be represented using more than one “channel”. For example, a computer monitor is trichromatic, because it uses red, green and blue (RGB) as the primary color channels. These colors channels are added together, each supplying an intensity value to represent the amount of that primary color in the overall mix of the displayed color. Much like the simple monochromatic images described above, precision is based off of the number distinct intensity values that can be represented per pixel. For example, in an RGB system with 8-bit precision per channel, each color can take on 256 distinct values and each pixel can represent $256^3$ different color values. The resulting system actually has 32-bit precision (or 32-bit color), versus 8-bit precision for a single color channel (monochrome) system.

Appropriate representation for image processing varies both by what is needed by the image processing algorithm and what kind of sensor equipment is available to capture an image. For example, if a monochrome camera was used to capture an image, it would make little sense to convert this image to a more complex trichromatic representation unless the image processing algorithm required this representation. No additional information could be gained by adding the extra color channels, since the image has only one channel to begin with. Therefore, many image processing algorithms work using a low-precision, monochrome image representation. The image can be reduced in precision to a single-channel monochrome image for processing even if advanced, high-precision image sensors are available.

The first step of image-aided navigation systems is feature extraction from an image. These features can be simple, such as corners and edges, or more advanced, such as landmarks, colors, or objects. Feature extraction is performed by an image processing algorithm. Typically, the complexity of the image processing algorithm increases in proportion to the complexity of the features being extracted, but the complexity of the extracted features is limited by the precision of the original image. For instance, a simple corner detection algorithm that finds all corners in a monochrome image could process images relatively quickly in comparison to a detection algorithm that detects all objects that contain four corners. However, complex features have more value in navigation applications because they are typically uniquely identifiable, making them suitable landmarks.

Features can be used for navigation in many ways. One of the most important ways to use them for navigation is through feature-correspondence algorithms. These algorithms work by transforming an image into its representative feature space. The features from an image taken at one point in time are matched to features from an image taken at a later point in time. The simplest way to perform feature correspondence is to perform a search for the features in one image over the entire space of another image, also known as an exhaustive search, illustrated in Figure 2.4. This procedure is computationally-intensive and time-consuming. For example, in a system with $N_k$ features in image $k$, the complexity of performing an exhaustive feature correspondence search for features in image $k+1$, in Big-O notation (Knuth 1976) is $O\ (\ N_k * N_{k+1}\ )$.

However, when other sensors are available, it helps to use these sensors to constrain the correspondence search space. In an image-aided INS, inertial measurements can be used to predict feature locations from one image to the next. The search area can then be constrained to a subset of the features in a successive image that are found within a certain distance from the predicted location. The resulting complexity is $O(N_k)$, which can be seen in figure 2.5. In addition, the difference between the predicted feature location and the actual feature location can be used to resolve IMU biases, a topic that is explained in further detail in Chapter 3.

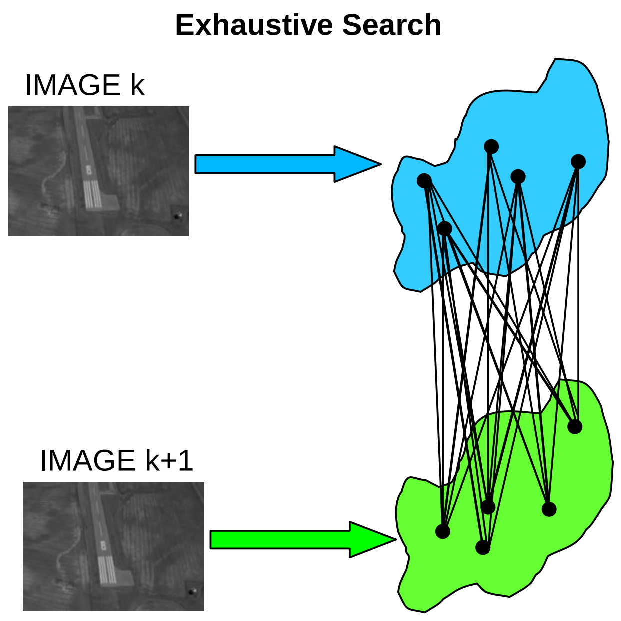


Figure 2.4: Exhaustive feature correspondence search. Extracted features are matched to each feature in successive images, resulting in $O\ (N_k * N_{k+1})$ search complexity.

Systems that employ feature matching have been of limited value in the past because most image processing algorithms were not suited to navigation purposes. Image processing for navigation requires features that can be uniquely identified despite movement of the sensor platform. The platform movement results in features being viewed from multiple angles, distances, and among other similar objects. To accomplish this feat, a feature has to have some kind of distinguishable identifier that is invariant to changes in orientation, scale and affine warpings. False identification is especially detrimental in feedback-enabled navigation, where the landmark data is used to correct other system sensors. Misidentification will corrupt the other sensor readings and lead to poor navigation predictions. Constraining the search space helps prevent false identification by eliminating a subset of possible matches. However, the main problem with using complex features from a computation standpoint is that feature extraction and correspondence requires extensive computation, which increases image processing time (Heymann 2007:6). Since images are used as an aid to navigation by correcting other sensor readings, a slow image processing algorithm contributes very little additional accuracy over the existing navigation sensors.

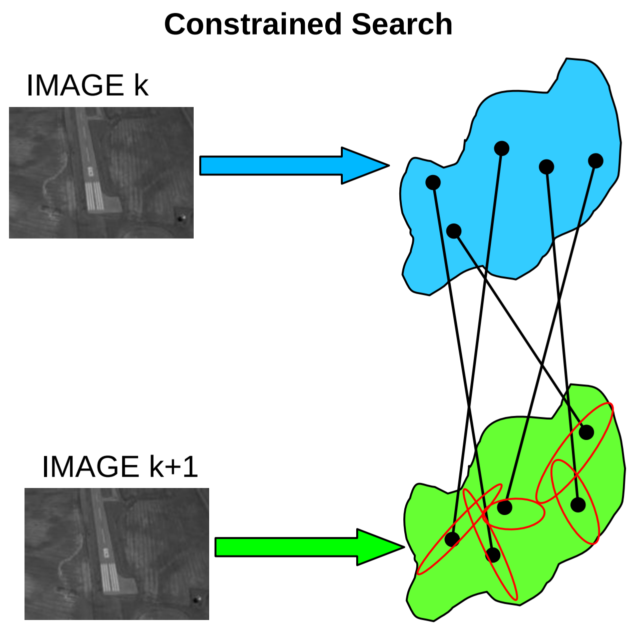


Figure 2.5: Constrained feature correspondence search. Features extracted from prior images are searched for in a constrained search space in successive images, resulting in $O(N_k)$ search complexity.

To overcome this problem and perform faster image processing, past systems have used *a priori* environment information to choose features. One such system, MOB-LAB, navigated streets by using lines on the road as features (Broggi 1995). The problem with this approach is that the environment becomes un-navigable when lines are unavailable, such as when the vehicle goes off-road. Other systems, such as Minerva (Thrun 2000) and (Panzieri 2003) tracked lights and ceiling tiles to determine their position at any epoch in time. Algorithms that use *a priori* information in this manner can be classified into a subset of navigation systems that rely upon reference landmarks for navigation. Such systems often use landmarks with known physical properties or fixed coordinates (i.e. waypoints) to help guide and correct the navigation system. However, to make an image-aided navigation system work in unknown environments, it must choose features that are invariant to scale and rotation without depending upon *a priori* environment information.

The feature extraction algorithm used in this thesis is the scale-invariant feature transform (SIFT) (Lowe 2004). This algorithm is used primarily for its complexity of features, which have a unique identifier that is invariant to scale and rotation. In addition, the identifier itself is not a complex data structure, so feature matching is straightforward and matching performance is high (Lowe 1999; Gordon 2004).

SIFT works by first decomposing an image into a scale-space representation. Decomposition results in a different type of information being available at each level of decomposition, but also allows the algorithm to separate extrema (local minima and maxima) in an image into the scale at which they are most prominent. Extrema are chosen using the Difference of Gaussian (DoG) from a progressively-scaled set of images that have been reduced to a lower level of detail via Gaussian blurring. Each extremum is chosen from the set of DoG images at the same scale, using pixel-wise comparison with each neighboring pixel at neighboring scales, but the same level of blurring. The resulting extrema are then filtered to remove points that do not have a high enough luminance value or that lie along an edge. These points are unfit for matching and correspondence. The resulting extrema are assigned a dominant orientation by taking a histogram of the gradient in a fixed area around each extremum. The gradient around the extremum is used in conjunction with the dominant orientation to compute a distinctive descriptor, which is invariant to rotation. A more in-depth look at SIFT can be found in many published works (Lowe 2004; Heymann 2007; Ke 2004).

Finally, an image-aided navigation system must take calibration into consideration. Image sensors, particularly camera lenses, suffer from image distortion that causes the captured image to look different than how it would look to the human eye. The type of distortion depends on the type of lens used, but is usually the barrel type of radial distortion, where the image appears to bow outward from the middle of the image (Draper 2002). This is caused by the hemispherical shape of the lens, normally referred to as a “fisheye” lens. Lens distortion can be avoided by using a rectilinear lens with no distortion, or corrected by undistorting the image. Image undistortion is performed by using the lens principal dimensions as well as focal and curvature parameters used to capture the distorted image, then using this information to map the hemispheric, distorted image, onto a 2D plane. This process is illustrated in Figure 2.6. Therefore, the degree to which lens distortion can be corrected is related to the precision and accuracy of the lens parameters used for undistortion. The process of determining these parameters is known as camera calibration.

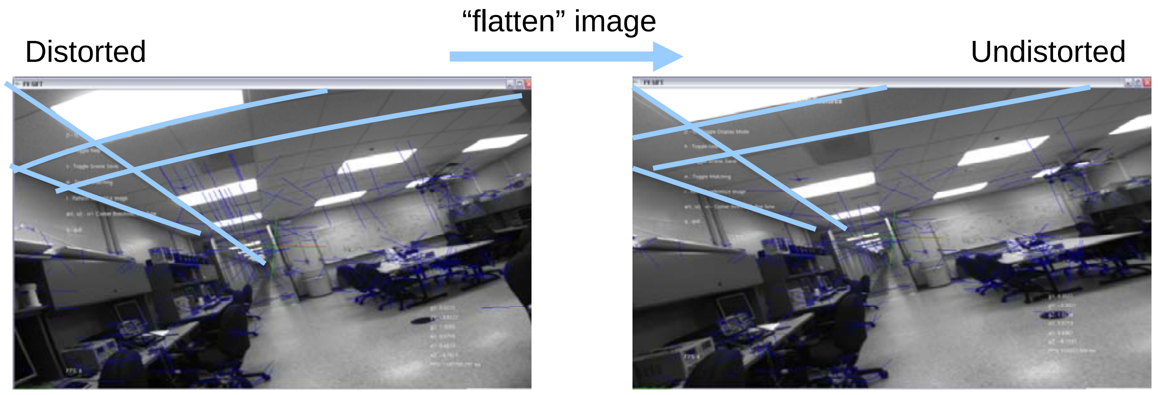


Figure 2.6: Camera image distortion removed. The left image exhibits noticeable curvature around the corners, also known as barrel distortion. The right image is the same image after lens calibration and image undistortion.

The calibration of an image-aided system can have a huge impact on the navigation predictions. The navigation predictions are corrected using the movement and disparity of features during feature correspondence. The feature correspondence algorithm is based on a flat image model, not a curved image. Therefore, when the features move on the curved, distorted image, their movement will not be translated correctly to movement of the sensor platform. This effect may not be noticed until the sensor platform changes position. To illustrate this point, take the analogy of two ants traveling in 2D space, shown in Figure 2.7. One ant is on a hill, the other ant is on the ground. They both start in the same position relative to an observer with a view from above. As the ants travel at the same rate, the ant on the flat plane appears to have moved further than the ant on the hill. This is analogous as what happens to features on the undistorted (flat) image versus the distorted (hill) image, as the sensor platform changes position. The features in an image will not appear in the predicted location during feature correspondence, resulting in higher measurement uncertainty and poor navigation performance.

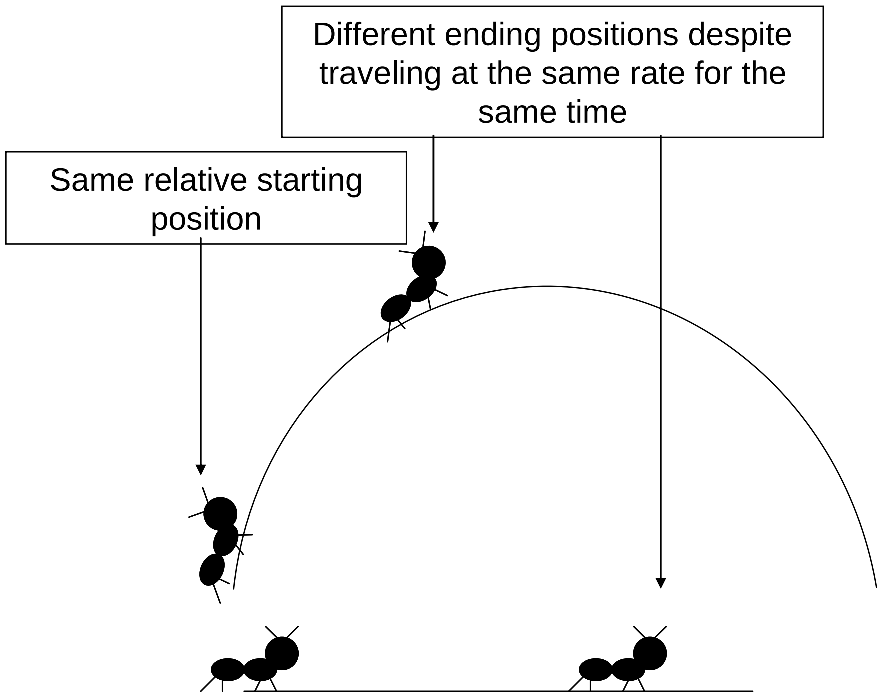

Different ending positions despite traveling at the same rate for the same time
Same relative starting position

Figure 2.7: Camera distortion analogy. The ant on the hill represents features on the distorted image, while ant on the ground represents features on the undistorted image.

### 2.1.4 GPS Navigation Systems

The Global Positioning Satellite (GPS) System provides civilian and military users with precise time and positioning information. This section provides fundamental information behind the GPS system, including a description of each segment and their function. The way to retrieve position information from GPS is described, along with the errors that accompany the position reading. Lastly, GPS modernization efforts are discussed so the reader knows which position errors will potentially be resolved in the near future.

GPS is divided into three segments: space, control, and user. The space segment consists of approximately 24 satellites that orbit the Earth. The satellites' orbits are arranged such that there are at least six satellites in line of sight from any point on Earth. As of April 2007, there are 30 satellites in orbit, mostly comprised of Block II-A through Block II-F satellites. The increased number results in a non-uniform arrangement, but provides redundancy and increased availability for GPS receivers. The research, development, launches, and maintenance of the satellite constellation costs approximately $750 million per year. The control segment consists of ground-based monitoring stations and a central control station that tracks the satellites, errors in their predicted path, and update settings in the satellites via a satellite uplink. The user segment of GPS consists of civilian and military users and their GPS receivers that receive GPS data from the satellites. A typical GPS receiver consists of an antenna and RF receiver, a stable clock, and a display to output speed and position information to the user.

Navigation data is sent from the satellite to the user via two microwave frequencies in the L-band; an L1 signal at 1575.42 MHz and an L2 signal 1227.60 MHz. Each signal is modulated by one or more codes which shift the carrier phase. These codes are each used for different purposes. The Coarse Acquisition (C/A) code modulates the L1 signal, and repeats a 1.023 Mhz pseudo-random noise (PRN) code every millisecond. There is a different PRN code for each satellite, so knowing the PRN for a specific satellite enables a receiver to pick out the satellite from which it is receiving data. The Precise (P) code modulates both the L1 and L2 signals. It is a much longer 10.23 Mhz PRN signal that repeats every seven days. Because of the length of this signal, receivers that have the capability and authorization to access the P code signal generally will acquire a satellite using its C/A code, then lock on to the P-code for further precision. The last code that modulates the signals is the navigation message, which is a 50 Hz signal consisting of data bits that describe the GPS satellite orbits, clock corrections, and other system parameters. This message repeats every 12.5 minutes and is modulated onto the C/A L1 signal. The relationship between the carrier signals and codes is shown in Figure 2.8.

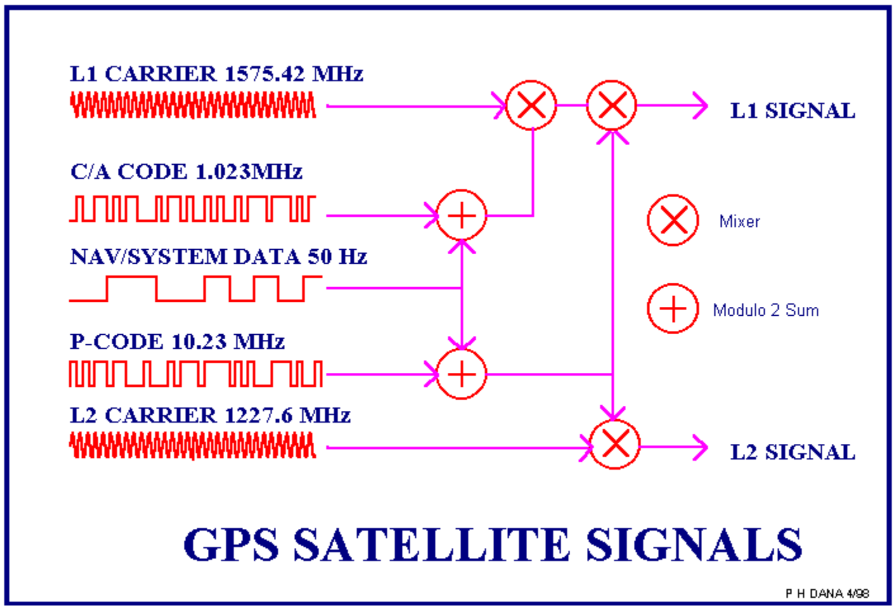


Figure 2.8: GPS signals and codes. 3 types of GPS codes (C/A, P, and navigation) are modulated onto carrier frequencies on the L1 or L2 band. Image from (Dana 2000).

The GPS receiver uses the navigation message from the satellites and an internal C/A code generator to determine the receiver's position. Since the C/A code is modulated onto the L1 frequency, the C/A code must be demodulated from the signal. An internally-generated PRN code sequence can be compared to the demodulated C/A code by shifting the PRN code sequence in time until a match is found with the PRN code from the received C/A code. The GPS Navigation Message consists of data bits time-stamped with the time of transmission by the satellite. The receiver uses the offset between the receiver clock and the GPS time from the navigation message to determine the time of arrival, or pseudo-range for an acquired satellite. Position is determined from multiple pseudo-range measurements at a single measurement epoch. The pseudo range measurements are used together with the satellite position estimates based on the precise orbital ephemeris data sent by each satellite. This data allows the receiver to compute the satellite position in three dimensions at the instant that they sent their signals. Four satellites are normally used to determine 3-D position and time, as shown in Figure 2.9.

However, there exist several potential pitfalls to GPS-based navigation that come from sources of error inherent to GPS. First, the GPS signal coming from the satellite is very weak. This means that the signal can easily be jammed or interfered with by non-GPS electromagnetic radiation at the same frequency. This interference can come from natural sources, such as electrical storms or solar flares, or from man-made signals. In the past, such sources have led to widespread GPS unavailability, and in worse cases, intentional GPS misguidance (Vizard 2003). In addition, the signal is so weak that it cannot penetrate buildings or the ground, making traditional GPS navigation indoors or underground currently impossible.

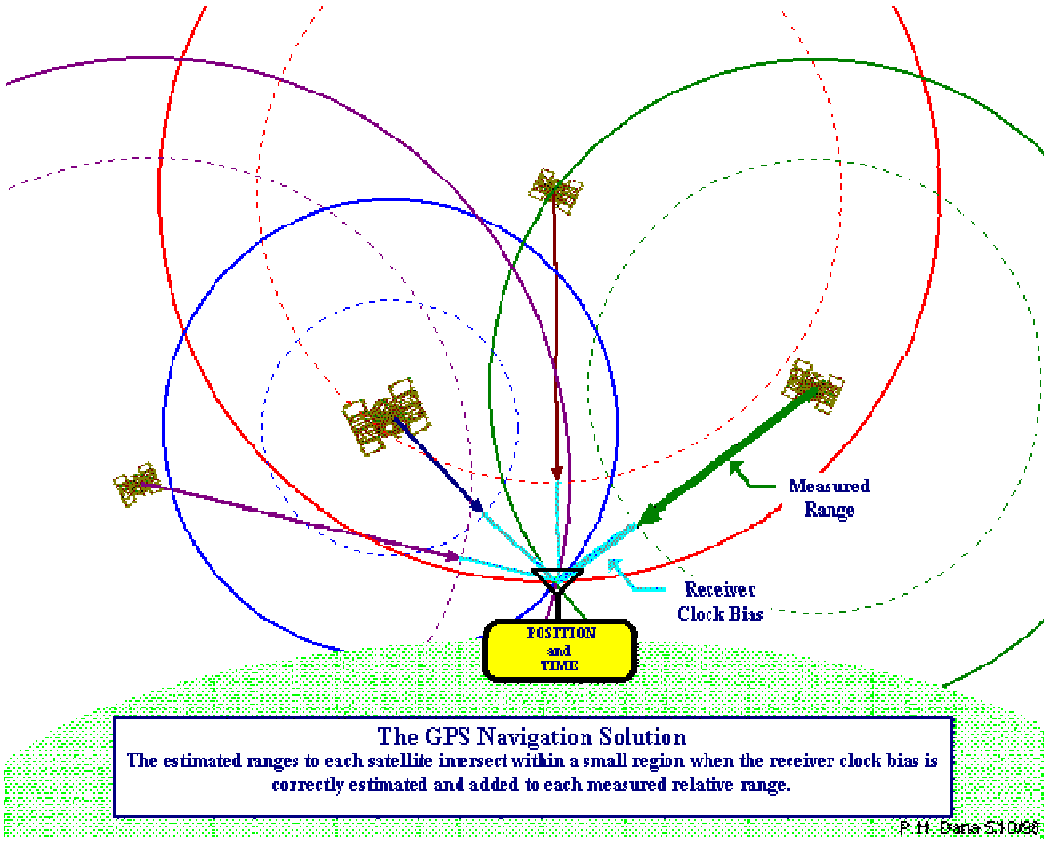

Measured Range
Receiver Clock Bias
POSITION and TIME
The GPS Navigation Solution
The estimated ranges to each satellite intersect within a small region when the receiver clock bias is correctly estimated and added to each measured relative range.
P. H. Dana 5/10/98

Figure 2.9: Position calculation using GPS. Using 4 GPS satellites, the GPS receiver finds the pseudorange and measured range to each acquired satellite to get a 3-D position and time information. Image from (Dana 2000).

Second, the GPS signal is subject to atmospheric interference. As the C/A and P/Y signals travel through Earth's atmosphere and ionosphere, the signal is affected by refraction. The receiver then calculates an incorrect delay from this signal, which in-turn negatively affects the position and distance calculations.  This condition is worse for satellites near the horizon.

Third, GPS is affected by a condition known as multipath, where the GPS signal bounces off of terrain in the environment before reaching the GPS receiver.  Multipath error also results in incorrect delay calculations by the receiver.  This condition is found more often in urban areas, where there are more structures that reflect GPS signals. Recent research into multipath problems in wireless networks has found that increased signal strength actually intensifies reflected signals (Ladson 2006)., so simply increasing the GPS signal power will cause additional multipath problems.

GPS modernization is currently underway.  Of the many improvements planned, the most influential for navigation are additional signals, frequencies, and improved signal strength.  These improvements will improve signal accuracy and alleviate some of the potential problems, such as atmospheric interference.  However, the problems of multipath and GPS unavailability indoors and underground will still exist, which provides more impetus for the exploration of GPS-alternative navigation techniques.

## 2.2 Biologically-inspired Navigation

The navigation system presented in this thesis is inspired by biological systems. Many animals have the innate ability to navigate naturally simply using biological sensors.  The problems with GPS are common to many other current navigation platforms, so several biological systems were researched in an effort to determine their viability for a GPS-alternative sensor platform.  Specifically, sensors that could work in a stand-alone environment, unaffected by interference and jamming were desired.

Some animals, such as sea turtles, use the static electro-magnetic fields (EMF) of the Earth to navigate (Lohmann 2001). By using the magnetic inclination angle and field intensity of these naturally-occurring signals, loggerhead sea turtles can navigate from the Eastern coast of the United States around the Atlantic Ocean, and back over a period of several years. The inclination and strength of the EMF signals provide the turtles with both a direction and a relative sense of position. However, scientists have found that by artificially duplicating the EMF signals, they can confuse the turtles' innate navigation ability and cause them to navigate as they would if the EMF signal was naturally occurring in the environment. Since EMF signals are abundant in the urban environment (Foster 2005) and are very easy to duplicate, any precision navigation system would have to account for the presence of artificial EMF signals and compensate accordingly. EMF-source detection and validation algorithms could be very complex and computationally costly. Therefore, EMF is not a good candidate for a GPS-alternative.

Ants and many other insects use chemical trails, such as pheromones and olfactory senses, to follow a path (Sharpe 1998; Marques 2002). These navigation systems work because paths that lead to food or safety are traveled more often, and the scents or pheromones become stronger along that path. This idea could be applied to GPS-alternative navigation by placing pheromone or other chemicals along a path in order to excite the olfactory senses during navigation for multi-agent systems. However, chemicals are subject to dissipation over time. In addition, most chemical signals can be replicated, so a malicious agent could fool the olfactory sensors by laying down a chemical trail to confuse or mislead the navigation system. Therefore, this method of navigation is susceptible to many of the same problems as GPS.

More elaborate biological navigation systems, such as those found in bats and dolphins, use sonar signals to get an estimate of terrain and distance. Bats are able to perform synthetic aperture sonar, which determines distance and direction information for all objects in the sonar “scene”, and reconstruct that objects shape (Griffin 1950; Simmons 2002). The shapes are then classified by the bat, allowing it to “see” an object and classify it. This sensor would therefore be extremely useful for a mapping application or landmark identification algorithms. However, sonar signals are susceptible to interference, jamming, and changing medium, and therefore exhibit the same weaknesses as GPS systems, making them unsuitable as a GPS-alternative sensor for navigation.

Animals that use passive sensors, such as bees, migratory birds and humans, however, are not susceptible to these problems. These animals all use inertial and imaging sensors to pick out landmarks in the environment and navigate from landmark to landmark with relative ease (Tripp 2001; Wehner 1996). In addition, these sensors are passive, self-contained and do not have the same weaknesses as GPS. Interestingly, desert ants also use this method of navigation in favor of pheromone navigation in unknown environments (Roumeliotis 2000). The most common way to perform image-aided navigation is to use visual sensors to pick out a landmark, or several landmarks, and navigate to that landmark, also known as waypoint navigation. The position between any set of landmarks can be determined using dead-reckoning techniques and the animal's inertial sensors.  However, if the animal's inertial sensors are imprecise, then the determined position will be poor.  Animals have overcome this problem by resetting their navigation state using a reference landmark with a known position (Knaden 2006).  If the animal can uniquely identify landmarks in its environment, then it can reliably determine its position in the environment relative to this unique landmark.  For instance, an ant will get lost quickly after leaving the nest.  However, if it periodically returns to the nest to reset its navigation state, the ant can navigate more precisely for longer periods of time.

The navigation system presented in this thesis is similar to that of landmark-following animals.  By choosing uniquely identifiable landmarks and determining the distance to these landmarks, a navigating body can determine how far a landmark should move in its field of view, based on the relative movement of the navigating body.  The relative movement of the navigating body is obtained from inertial motion sensors, much like those present on the animals described in this section.  The landmark movement can thus be used to correct inertial readings and corrected inertial readings can be used to aid landmark location predictions.

## 2.3 A Brief Introduction to the Graphics Processing Unit

The following section provides background information on the Graphics Processing Unit, or GPU.  A goal of this thesis is to harness the power of the GPU to perform image processing needed for navigation computations.  To understand how this can be accomplished, one must be familiar with fundamental GPU concepts.  This section provides information on how the GPU can be used to perform tasks normally performed by the CPU.  GPU data processing flow is fundamentally different than the CPU, so the basics of the GPU rendering pipeline is covered to help the reader understand the flow of information through the hardware. Programming models, including high-level shading languages and low-level assembly are briefly covered. Lastly, performance data for GPU-accelerated tasks is reviewed so the reader has a baseline for expected GPU acceleration for the navigation system in this thesis.

### 2.3.1 General-Purpose Computation using the GPU

The GPU was invented because graphics-intensive applications such as video games and computer-aided design (CAD), required increasing numbers of complex calculations over large sets of uniform data types, and these applications were experiencing a bottleneck in CPU processing power (Trancoso 2005). To overcome this bottleneck, manufacturers had a choice of building faster CPUs to handle the computational workload, or to build a hardware graphics co-processor that could perform specialized calculations on the graphics data. The GPU thus evolved as a hardware co-processor to perform calculations on large sets of uniform graphics data. The CPU, on the other hand, remained suited for a multitude of different calculations on elements of varying data types. Over the years, the GPU became faster, the processing pipeline became more customizable, and the instruction sets became more diverse. These improvements were originally intended for richer graphics, but the idea of performing computation on data beyond the graphics primitive domain on the GPU received increasing attention. The idea of extending GPU computation beyond the graphics domain became known as general-purpose computing on the graphics processing unit, or GPGPU (Thompson 2002).

There are three major reasons to use the GPU for general-purpose computing (Metelitsa 2005). First, the GPU is commodity hardware and comes standard on almost every commercial off-the-shelf (COTS) computer sold today. This comes with a huge advantage in price / performance ratio, since R&D and manufacturing costs for equipment are constrained to the commercial market. Secondly, although the clock rate of GPUs are lower than CPUs, the parallel architecture of the GPU and the locality and speed of GPU memory results in an overall throughput rate that is much higher than a CPU, and increasing at a rate faster than Moore's Law (Moore 1965). Lastly, since the GPU acts as a co-processor, computations that are performed on the GPU free up the CPU to perform other tasks.

GPGPU also has a few drawbacks. First, the architecture and processing flow of the CPU and GPU are fundamentally different. The CPU acts as a multiple-instruction, multiple-data, single element processor. It has a wide set of instructions for a wide range of data types, but performs sequential processing, or computations on a single element at a time. The GPU acts more like a single-instruction, multiple-data, stream processor. It has a limited range of instructions for a very limited set of data types, but can perform these computations in parallel on a large set of data, known as a stream. This fundamental difference is best illustrated by a pipe-and-filter architecture (Garlan 1994), as shown in Figure 2.10. Although this design is a strength of the GPU, it introduces a challenge to the GPGPU programmer: namely, how to adapt from the sequential processing model of the CPU to the stream processing model of the GPU.

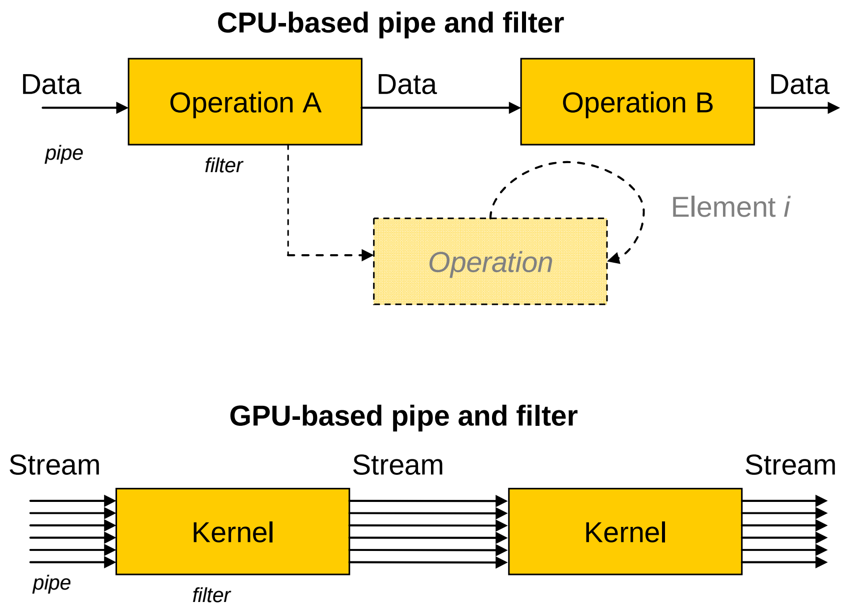


Figure 2.10: A comparison of GPU and CPU-based pipe and filter architectures. The CPU must perform operations upon data elements in sequence, whereas the GPU can act upon Stream elements in parallel.

### 2.3.2 GPU Rendering Pipeline

The GPU rendering pipeline, which creates a 2D image from 3D geometry and textures, has evolved from a fixed-function implementation to the current fully-programmable pipeline. The original goal of GPU manufacturers was to maximize throughput of data elements to the screen. The goal has not changed, but manufacturers are now also focusing on building in as much programmability as possible without compromising performance (NVIDIA 2007).

To understand GPU rendering, one must first understand the components of a GPU program. A GPU program consists of commands, vertices, fragments and textures.

Vertices are points in 3D space that are connected to make up geometrical objects, such as lines, triangles, and polygons. Fragments are the equivalent of pixels on the screen, complete with color and location information. Textures are images that can be projected onto a geometrical object. Some authors have described textures as something like "shrink wrap" (Metelitsa 2005) for geometry. These textures are applied by performing a texture lookup, which changes the color value of a set of fragments based on the size, shape, and color value of the texture. GPU commands specify the connection order of vertices, which textures to apply, and which vertex and fragment shader programs to load. All of these components are connected together in the programmable GPU rendering pipeline.

The programmable GPU rendering pipeline begins in the CPU. The CPU must interact with the GPU via the video card manufacturer's drivers and the hardware API used to interface with those drivers. The application specifies the 3D geometry and textures to be displayed on the screen. This data may or may not be stored in system memory for future use, depending on the application. This information is sent as vertices to the GPU over the system bus, and then placed in high-speed memory resident on the GPU. The GPU retrieves the data placed in video memory and computes a 2D image of the geometry using vertex processors and rasterization. This process can be customized using a vertex shader program. The GPU then computes the appropriate color for each fragment using a texture lookup, fragment shader program, or a combination of the two. The resulting 2D image can either be displayed to the screen by writing to the GPU frame buffer, or saved for future display (or another rendering pass) by writing to an off-screen pixel buffer. This rendering loop is illustrated in Figure 2.11.

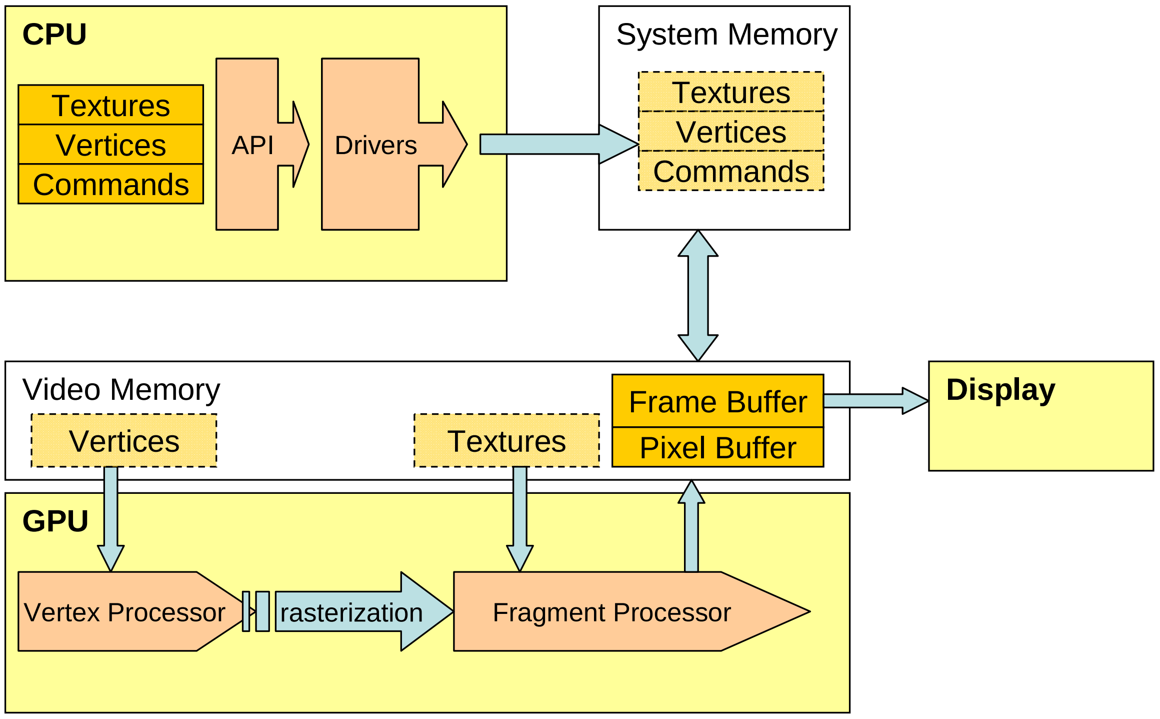


Figure 2.11: The GPU rendering pipeline. Textures, vertices, and commands are created in the user program, interpreted by the hardware drivers and sent into main memory. The vertices are sent to the vertex processor, rasterized to a 2D graphics representation and sent to the fragment processor.

### 2.3.3 GPU Programming Languages

As mentioned previously, the GPU and CPU architecture and hardware are designed for different purposes. Programmers must understand and account for these differences when writing GPGPU programs. Although the GPU is capable of very powerful operations, the type and number of operations are limited in comparison to the CPU. For instance, the input and output stream size and precision, as well as the temporary registers and the number of instructions are all limited by the GPU hardware. In addition, there is no jumping, looping and very limited branching native to GPU programs. These limitations have been compensated for by using multi-pass rendering and high level programming languages that can emulate branch and loop behavior, but at a high rendering cost. This section provides an overview of some of the more common GPU programming techniques. An illustrated overview of the discussed techniques can be found in Figure 2.12.

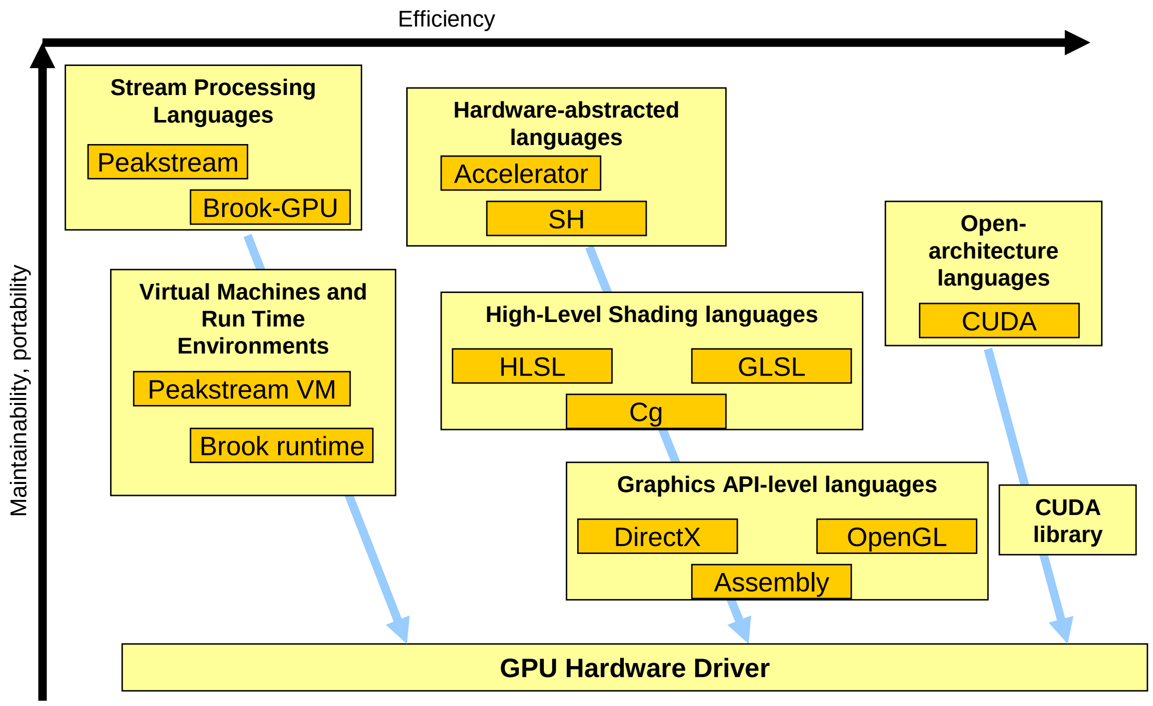

Efficiency
Maintainability, portability
Stream Processing Languages
Peakstream
Brook-GPU
Hardware-abstracted languages
Accelerator
SH
Open-architecture languages
CUDA
Virtual Machines and Run Time Environments
Peakstream VM
Brook runtime
High-Level Shading languages
HLSL
GLSL
Cg
Graphics API-level languages
DirectX
OpenGL
Assembly
CUDA library
GPU Hardware Driver

Figure 2.12: Hierarchy of GPU programming tools. Although hardware abstraction via language extensions and generalized stream processing languages increases portability and maintainability, performance and efficiency suffer.

Much like the CPU, the most computationally-efficient way to program a GPU is by programming in assembly language using the application's graphics API. Typically, video card manufacturers implement both the OpenGL and DirectX graphics APIs in the GPU hardware driver. Programs can either be passed as a character string to the 3D graphics API or loaded as an object. Data is operated on as a texture, functions are performed on an entire set of vertices or fragments using the GPU rendering pipeline, and multiple rendering passes are performed using copy-to-texture or render-to-texture operations. The basic tenets of GPGPU programming using the GPU hardware APIs can be found in (GPGPU 2007). While programming in assembly language may be the most computationally-efficient, it is time-consuming, mundane and error-prone for the programmer. In addition, much like programs written in CPU assembly, programs written in GPU assembly lack usability, maintainability, and portability (Metelitsa 2005).

Another efficient way to perform GPU programming is by using a high-level shading language, such as Microsoft's HLSL, the Open Standards Group's GL shading language (GLSL), and Nvidia's C for Graphics (Cg). These languages provide programmers with a high-level syntax, operations and data structures for simplified programming, similar to what high-level languages like C did for CPU assembly language. However, each of these high level languages requires a graphics API for programming. HLSL depends upon DirectX, GLSL depends upon OpenGL, and Cg can be compiled to use DirectX, OpenGL or the Cg API.

The major disadvantage of the high-level shading languages is that the programming model is tailored toward graphics processing rather than general-purpose computation. These languages are limited to performing operations on textures, vertices, fragments, and other data structures in the graphics hardware domain. However, there are a few hardware-abstracted languages that have been created for efficient stream processing and general computation using the GPU. Most of these languages merely provide the programmer with automated shader program creation. These languages encapsulate the high level shading languages, such as GLSL or HLSL, to provide an interface to the GPU hardware layer. Other hardware-abstracted languages are more efficient because they call into the graphics API to create custom shader programs. These languages are usually implemented as a library or extension to the application's programming language that wrap GPU API calls. Examples are the University of Waterloo's SH (McCool 2004) for C++ and Microsoft's Accelerator (Tarditi 2005) for C#.

Another approach for GPGPU programming is to create a new language with a new runtime environment. Two platforms take this approach; Stanford's Brook GPU language (Buck 2004), which is an extension to C, and Peakstream (Peakstream 2007), a new project sponsored by ATI. Brook uses programs written in the Brook GPU language and ANSI C and the Brook runtime to create customized stream programs. The downfall of this technique is that the resulting stream programs are not very efficient, since the shader commands are not tailored to any specific hardware driver, platform, or customized for the type of GPGPU program being written. The Peakstream language uses a virtual machine (VM) approach, much like Java, to make a very portable solution that can work across many platforms and types of hardware (Peercy 2006). Since this project is still in development, initial results have shown only a modest speedup from GPU acceleration. However, the Peakstream project has great potential, since the VM can be updated and modified for future platforms and video hardware, and support more powerful GPU commands that would increase the efficiency and performance of the resulting programs.

Recently, NVIDIA has produced a new GPU with an open architecture that is built as both a graphics and GPGPU powerhouse. This hardware is coupled with a new GPGPU programming SDK called CUDA. CUDA gives the GPGPU programmer the ability to read and write any area of video memory while at the same time abstracting the hardware from the programming language. The result is that a programmer using CUDA can program in much the same way as they would on a CPU, but perform their operations in parallel using GPU hardware (NVIDIA 2007). In addition, the GPGPU API is supported natively by the hardware, so CUDA commands do not need to be interpreted by the hardware-layer API like OpenGL or DirectX, making CUDA more efficient than the current generation of GPGPU APIs.

### 2.3.4 Performance

GPU and CPU performance can be compared in many ways. One of the most basic comparison metrics is the measure of floating point computations possible per second (FLOPS). Since most modern processors are capable of billions of FLOPS, the performance metric often used is gigaFLOPS (GFLOPS). Fortunately, there is a benchmark available to compute just that; the Linpack benchmark (Dongarra 2001). The Linpack benchmark is a measurement of the performance of a dedicated system in solving a dense system of linear equations. The test is reliable, repeatable, and commonly accepted as a valid measure of peak computation performance. More importantly, the test results can be used to compare the general computation ability between different computers and different computing architectures. The comparison of GFLOPS ratings between some common NVIDIA GPUs and leading Intel CPUs is shown in Figure 2.13.

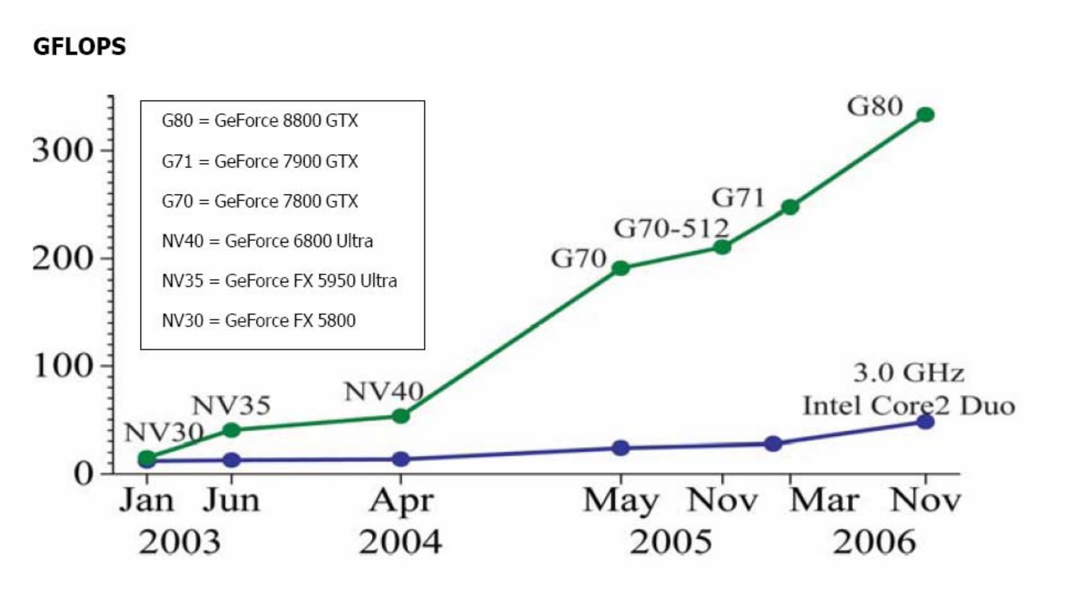


Figure 2.13: GFLOPS comparison between GPU and CPU. The GFLOPS rating of CPUs is growing at the pace of Moore’s Law, while GPU ratings are growing much faster. This is due to superlinear speedup achieved through parallel computation (Akl 2001). Image from the Nvidia CUDA programming guide (NVIDIA 2007).

Knowing that the GPU computational performance should be much greater than the CPU performance, the next logical step is quantification of how much better the GPU performance should be. The simplest way to quantify this performance is through a term known as speedup. Speedup is the difference in computation time between the original process ($T_1$) and the new process ($T_2$), as shown in equation 1.

$$Speedup = \frac{T_1}{T_2} \tag{1}$$

The metric used for speedup comparison is SIFT feature extraction time. The computation time to extract SIFT features can be compared for images with different numbers of features and different resolutions. The baseline SIFT performance data in Figure 2.14 was taken from Sudipta Sinha's implementation (Sinha 2006). The results lead to an expected speedup of 6x-15x for GPU-accelerated feature extraction.

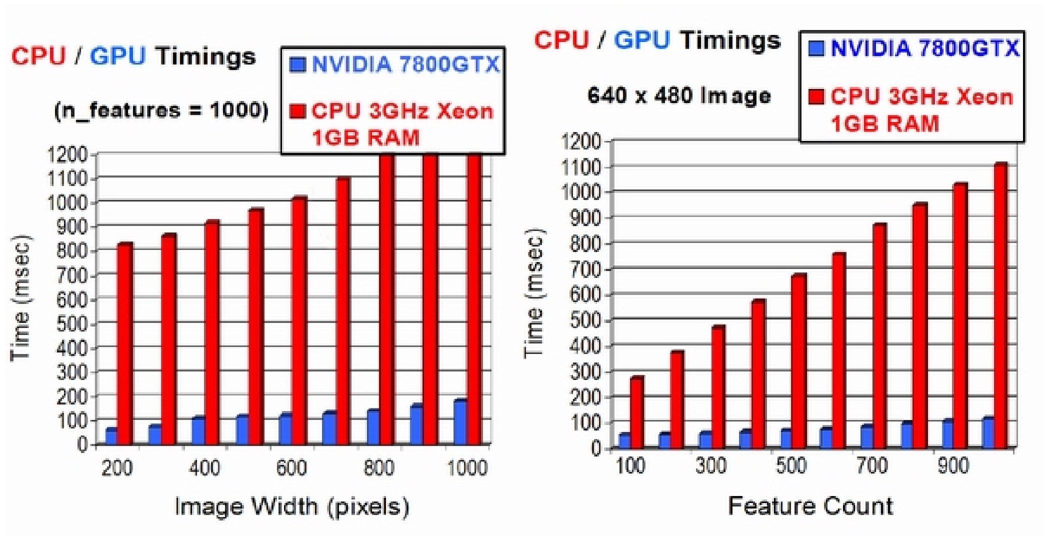


Figure 2.14: Comparison of CPU and GPU SIFT feature extraction time. Expected times based on GPU-SIFT program by Sudipta Sinha. Image courtesy of (Sinha2006).

A 2002 study by Thompson (Thompson 2002) of GPGPU techniques was used for the speedup baseline shown in Figure 2.15. The speedup for CPU vs. GPU matrix multiplication can be expected to be a function that increases linearly with matrix (or image) size.

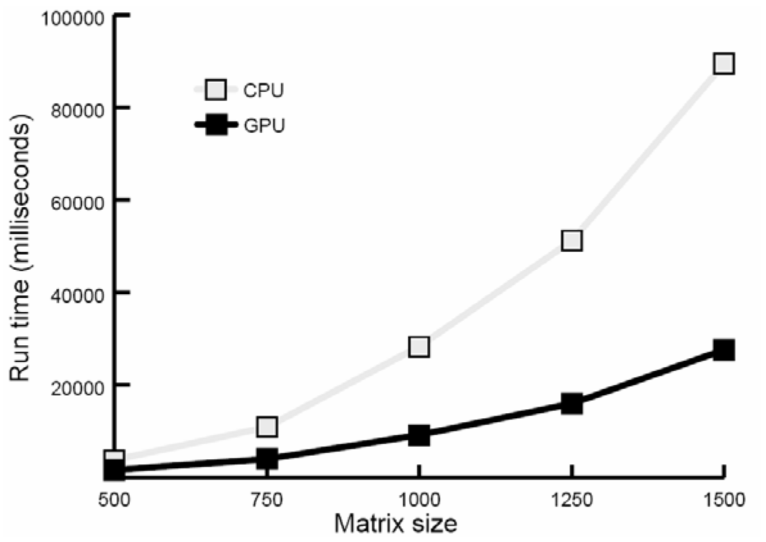


Figure 2.15: Expected runtime for GPU vs. CPU matrix multiplication. As expected, the CPU run time increases exponentially with increasing matrix size. However, GPU run time increases linearly. Graph from a survey of GPGPU techniques (Thompson 2002).

Despite the method used to acquire and quantify performance data, the expected results may differ from the actual results. Actual performance and speedup is a result not only of the algorithm being run, but the speed and efficiency of the hardware, software application, and underlying operating system. In addition, the speedup from GPU-accelerated image processing is only a fraction of the overall navigation system. To take advantage of this speedup, there must be no bottlenecks in GPU or CPU processing in other parts of the system.

## 2.4 An Overview of Real-time Processing

One of the goals of this thesis is to create a system that could navigate in real-time. To understand what was needed to accomplish this feat, one must understand what it means for an application to run in real-time. This includes historical and modern definitions of a real-time system. In addition, when designing an application for real-time operation, the system must be categorized as a hard or soft real-time system and the corresponding constraints must be taken into consideration.

### 2.4.1 History of Real-time Processing

Real-time processing has been redefined many times over the past few decades. The term originally referred to any system that could perform computations at a rate that matched or exceeded that of the real process it was simulating (Heitmeyer 1996). This definition changed during the advent of thread priority scheduling and programmable microcontrollers. These technologies allowed a process to pre-empt the operating system and schedule a process to run for an arbitrary amount of time. Precise thread scheduling resulted in increased reliability and predictability, since the application could be given a time constraint in which it must complete its task. However, if the task was not completed in time, the application could either fail or continue, which led to two separate schools of thought for real-time process scheduling.

Current real-time systems are now divided into two categories; hard real-time systems and soft real-time systems. Both categories are based on constraining a task using a deadline from an event to a system response. The two differ in that a hard real-time system considers the system response after a deadline to be useless, whereas a soft real-time system will tolerate a missed deadline (usually with some impact on performance), and continue to operate (Juvva 1998). Hard real-time systems are thus best suited to safety-of-life applications, while soft real-time systems are suited to applications that involve concurrent access with changing situations.

### 2.4.2 Real-time Constraints

Hard and soft real-time systems can be considered constraint-based systems, where the constraint is the time from event to system response. A hard real-time system must complete its task in the constrained timeframe or fail. A soft real-time system must complete its task in the constrained timeframe or suffer degraded performance. However, the type of constraint (hard or soft) does not necessarily relate to the length of time available to complete its task. In other words, some hard real-time systems have a longer deadline than soft real-time systems.

Deadline calculation is usually application-specific and depends both on the system constraint type (hard or soft) and the desired level of performance. For instance, in an application such as online game play, the goal is to minimize ‘lag’, which is the time between when a user enters a command and the online players are updated with the effects of that command. The desired level of performance is instantaneous effect, or zero-lag, which is unachievable, because the deadline is affected by time-consuming factors such as network speed, command processing time and the time necessary to update online users. The deadline can be computed by the network speed and the time it takes for the game to propagate effects from a command to the players. If the game had a real-time deadline, it could perform one of two operations if the deadline was exceeded. The game could fail, by disconnecting the players and exiting, making it a hard real-time system. On the other hand, the game could tolerate the missed deadline by pausing game progress until all players had been updated. This degraded service in a soft real-time system is preferable for this system example.

However, the deadline for an anti-lock brake system might be computed in a much different way. The anti-lock brake system on a car is designed for safety-of-life, which typically dictates a hard real-time system. The deadline would be computed using the stopping distance of the car, the effectiveness of the brakes, and the anticipated reaction time for the driver. Designers will typically first establish the constraints for a hard real-time system, then design the system to meet those constraints.

Additionally, real-time systems can utilize a dynamic deadline (Kopetz 2002). Such systems are characterized by a non-fixed constraint time from event to system response. Take the online game mentioned previously as an example. The deadline can be determined by timing the network speed as well as the application response time. However, if the game is played over a slower network, or if a player on a slower machine is connected, the deadline could be adjusted longer. The longer deadline results in additional lag, but the system will behave more predictably. Dynamic deadlines are also useful for multi-sensor platforms. The deadline for a real-time system could be adjusted for the acquisition rates of the different sensors, which may be added, enabled, or disabled while the system is running.

### 2.4.3 Real-time Scheduling and Design

Real-time systems are often managed by a scheduler. The scheduler can be either pre-emptive or non pre-emptive. The pre-emptive scheduler will interrupt a process if it has not completed its task by the deadline. A non pre-emptive scheduler will allow a task to run to completion before scheduling a new task, so processes are never interrupted. In addition, a scheduler also has a priority scheme that is used to schedule processes. Some common scheduling algorithm are earliest deadline first (EDF) and least laxity (LL). Although these scheduling algorithms have been proven optimal on single-processor computers, they have been proven sub-optimal on multi-processor systems (Kopetz 2002), due to task priority issues.

Whether the system is hard or soft, pre-emptive or non pre-emptive, one of the goals is always to minimize the response time of the system. The response time is governed by the choice of system. For example, the shortest guaranteed response time for a non pre-emptive system is the longest task time plus the shortest task time (Kopetz 2002). Therefore, when designing a real-time system, one must always keep the response time factor as a priority and choose appropriate real-time system constraints and scheduling system. This thesis was concerned primarily with non-pre-emptive, soft real-time constraints, which consisted of minimizing the time between the event and system response for minimal latency.

## 2.5 Object-oriented Software Design Principles

The navigation system presented in this thesis was designed as a component-based object-oriented system. Object-oriented design (OOD) can be very complicated, and there are many software engineering tools and processes that can be used for OOD. One such method is the rational unified process, or RUP, which evolved as an improved spiral model (Larman 2005) process. The heart of RUP is comprised of six key principles, summarized in Table 2.1.

Table 2.1. Six key principles of the Rational Unified Process

| Process Name | Description |
|---|---|
| Adapt the Process | The OOD process should fit the organization, adapt it as necessary. |
| Balance Stakeholder Priorities | Project goals should be a balance of what the software designers want and what the business goals are. |
| Collaborate Across Teams | Communication with the other software architects on the project is not just expected, it is essential. |
| Demonstrate Value Iteratively | Deliver projects iteratively, rather than having nothing concrete from start to finish except the final product. |
| Elevate the Level of Abstraction | Use software patterns and frameworks for code resuse. |
| Focus Continuously on Quality | Perform quality checks and automated tests at each iteration. |

These principles are used to guide decisions made during the RUP phases. The RUP consists of four distinct phases; inception, elaboration, construction and transition. Inception consists of meetings with stakeholders and discussion of requirements, usually resulting in the initial budget, schedule, and a set of use cases that diagram the desired interface and functions of the software. The elaboration phase is where the project begins to take shape. Most of the use cases are defined during elaboration, as well as an identification of the major risk areas. In the construction phase, software engineers develop the major components of the software, starting with the highest-risk features. Lastly, in the transition phase, the software is transformed into a state that will be usable by the end-user.

Another important consideration during software design is the software architecture that is used. The software architecture can be considered the skeleton or framework of the system. Just like there are many types of skeletons in the world, there are many varieties of architectures available for software design. These architectures can also be composed and adapted for specific problems. Since the real-time system architecture consisted of multiple-sensors acting on an underlying system state, the decision was made to use the model-view-controller (MVC), illustrated in Figure 2.16 and blackboard architecture, illustrated in Figure 2.17.

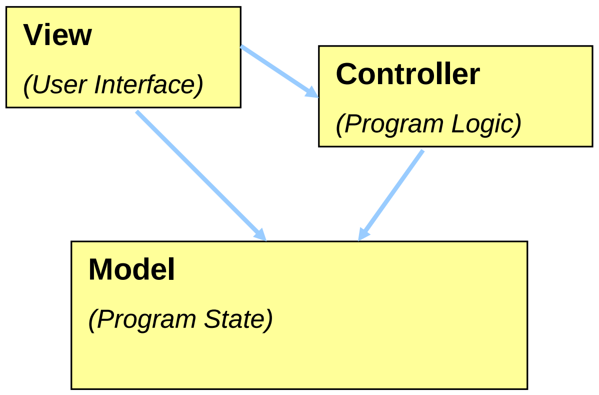


Figure 2.16: Model-view-controller architecture. Lines indicate visibility of lower layers. Higher layers can also affect change to lower layers.

The MVC architecture is used for separation of concerns, so that software components can be changed or removed with little to no effect on the rest of the system.

This is accomplished by layering, where higher layers have knowledge of and can affect changes to the lower layers, but not vice-versa. The model component, accessible by the controller and view components, contains the program state, consisting of variables and data structures. The controller typically drives changes to the model, based on inputs from the view component and any control policies defined at a system level. In addition, if the software system needs to access system hardware, such as attached sensors, the hardware control routines are often built into the controller component. The view component typically consists of the user interface and any user command parsing, display routines, and menus needed for user interaction. The view component is often operating-system dependent, since most operating systems have a built-in windowing system.

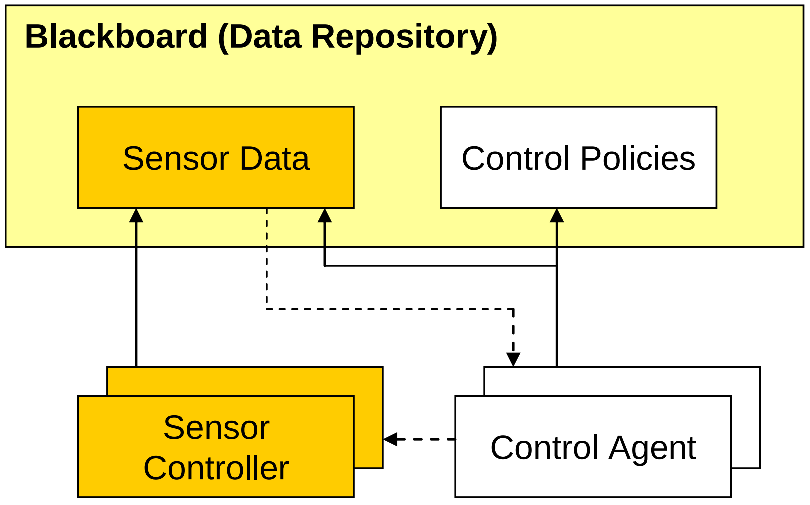


Figure 2.17: Blackboard Architecture for multi-sensor, multi-agent systems. Control policies and sensor data is stored in the blackboard, which is shared between multiple sensor controllers that write the sensor data, and control agents that use the data and set control policies.

The blackboard architecture was chosen because it is typically used to decouple a multi-sensor, multi-agent platform. The blackboard architecture uses a central data repository (blackboard) object to which many sensors can independently write. The blackboard can then be accessed by control agents that use the sensor data. In addition, the blackboard can be used to hold access policy information, which dictates the behavior of the control and sensor agents. A similar multi-sensor fusion architecture has been applied to the problem of Network-Centric Warfare, which requires integration of multiple sensors while performing persistent data retrieval (Landing 2006). The sensor acquisition problems and architecture solutions are similar between these systems, which help to validate the decision to use the blackboard architecture for this navigation problem. The blackboard architecture also integrates well with the MVC architecture, since the blackboard can act as the underlying model, while sensors and control agents are controller objects that access and change the model. This relationship is illustrated in Figure 2.18.

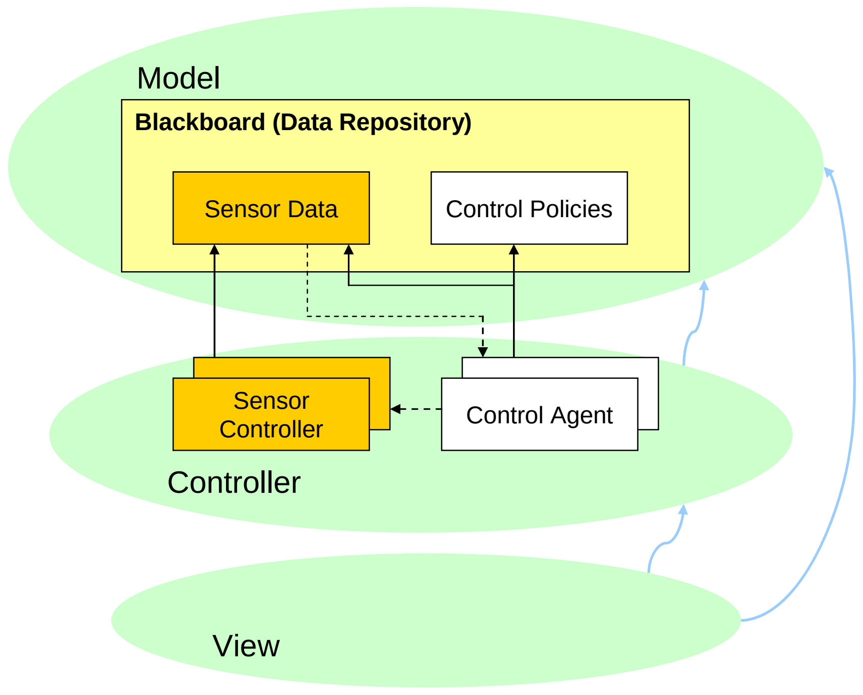


Figure 2.18: MVC Architecture integrated with blackboard architecture. Sensor data and control policies are stored in the model, or state, of the system. Data is shared between the sensor controllers and control agents.

## 2.6 Summary

Many fundamental concepts were presented in this section. First, the idea of precision navigation using an inertial navigation system and image-aiding were presented. The GPS navigation system and its flaws were briefly mentioned. The next topic covered was the concept of biologically-inspired navigation systems, and which system worked best as a GPS-alternative navigation system.

The next topic covered was general purpose computation on the graphics processing unit, or GPGPU. The fundamentals of GPU processing, programming, and rendering were covered. In addition, the problems with porting CPU-based sequential programs to GPU-based stream programs and alternatives were explored. The GPU section concluded with expected speedup values for GPU acceleration for feature extraction.

The next topic of interest was real-time processing. The history and definition of real-time processing was first presented, followed by information on more modern definitions of real-time processing, including deadlines and real-time constraints. The challenges associated with real-time processing were also mentioned in the context of the material presented in this thesis.

Lastly, the topic of software design principles was covered. The majority of the design principles were about multi-sensor integration using the MVC and blackboard architecture. Software design patterns and the RUP were also covered, as these design elements were utilized during the software engineering section of this thesis. The software engineering techniques employed in this thesis were paramount to the overall speedup over the existing navigation system.

The intent of this section was to provide the reader with a theoretical background to understand the challenges in the software and hardware design of the navigation system presented in this thesis. It is a challenge just to get an image-aided inertial navigation system to function. It takes considerable effort to get such a system to work in real-time.

# III. Methodology

The purpose of this chapter is to describe the design of the navigation system presented in this thesis. The navigation system design is divided into two primary areas; hardware and software. In the hardware design sections, the overall concept and goal of the hardware design is discussed, followed by the interface and component choices. Additional issues of power consumption and component synchronization are evaluated, resulting in the final design. The software design is approached from the software engineering standpoint. The use of software engineering design principles, such as the RUP and design patterns, is discussed. The details of the implementation are presented from the standpoint of efficient, concurrent, real-time design.

## 3.1 Hardware Design

The hardware used for this thesis was designed for two purposes; to perform concurrent capture and image processing, and to allow for real-time navigation. The computing platform and sensors were chosen such that they could be controlled via a real-time scheduler, so data would be immediately available for use in the real-time navigation component. In addition, three principles drove hardware design for this thesis; mobility, compatibility, and precision. This had an effect on what IMU, image sensor, computing platform, and power source was selected for the system.

As mentioned in chapter 2, an inertial measurement unit is a trade-off of precision for mobility. The tactical-grade Honeywell HG-1700 IMU, for example, measures 33 cubic inches and 726 grams. It has a maximum gyro bias of 10 degrees per hour. However, the consumer-grade Crista IMU that was used in the system is only 3.2 cubic inches and weighs 37 grams, but has a maximum gyro bias of 36 degrees per hour. Table 3.1 reviews some common IMUs and their specifications from their manufacturer's technical data sheets. The Crista was chosen because mobility was more important than precision, since the inertial sensor was being corrected by the image sensors.

Table 3.1. Common IMU specifications

| **IMU** | **Size (in3)** | **Weight (grams)** | **Gyro Bias (deg / hr)** | **Random Walk (deg / rt hr)** | **Drift (nmi / hr)** |
|---|---|---|---|---|---|
| Crista | 3.2 | 37 | 12 - 36 | 6.48 | [1] |
| MIDG II | 2.2 | 55 | [1] | 1.7 | [1] |
| HG-1700 | 33 | 726 | 1 - 10 | 0.125-0.5 | [1] |
| HG-1900 | 20 | 454 | 0.3 – 10 | 0.1 | [1] |
| HG-9900 | 103 | 2951 | .003 | .002 | 0.8 |

[1] data unavailable

The image sensors chosen for the hardware design were IEEE 1394 (Firewire) cameras. The Firewire interface provides a sustained data transfer rate of 400 Mbps, which is a good backbone for data communications with the computing platform. By contrast, the Universal Serial Bus (USB) 2.0 interface provides a sustained data rate of 480 Mbps. However, most USB cameras are not capable of the high-resolution imagery that Firewire cameras are capable of. In addition, the IEEE 1394 camera specification (IEEE 1998) defines a common driver architecture called DCAM, which vendors can optionally implement. What this provides the programmer with is a common API that can be used to run any DCAM-compliant camera connected via the IEEE 1394 bus, which provides "plug and play" sensor capability. Firewire cameras were chosen for the image sensors due to the possibility of a unified interface, enhanced compatibility, and precision.

The computing platform was chosen because of its compatibility, mobility, and capability of high-precision computation. One of the hypotheses for this thesis was that the image processing step of image-aided navigation could be offloaded to the GPU, which necessitated the use of a system that contained a GPU. There were many desktop computing workstations available, but a laptop computer is much more mobile. The computing platform choice was a trade-off of computation ability for mobility. For instance, a quad-core processor machine with two parallel video cards could have been chosen, and probably would have handled the task very well, but one of the goals is to get the navigation system portable enough to be used on vehicles, which requires a smaller form factor. Two laptop computers were acquired and used for the computing platform. The first was a small, light, ultra-portable business laptop containing a hybrid, hardware-integrated GPU. The second was a larger, more powerful, yet still very portable gaming laptop. The specifications of each system are laid out in Table 3.2.

Table 3.2. Computing Platform Comparisons

| **Mfg** | **Model** | **CPU** | **GPU** | **GPU Stats** | **RAM** | **Wt (lbs)** |
|---|---|---|---|---|---|---|
| Sony | SZ220 | T2400<br>1.83 GHz<br>2 MB cache | NVIDIA<br>GeForce Go<br>7400 | 128 MB shared<br>3 : 4 : 4 : 2 [1] | 1 GB<br>DDR2<br>533 MHz | 4.07 |
| Dell | XPS1710 | T7200<br>2 GHz<br>4 MB cache | NVIDIA<br>GeForce Go<br>7950 GTX | 512 MB<br>8 : 24 : 24 : 16 [1] | 2 GB<br>DDR2<br>667 MHz | 8.7 |

[1] Vertex Shader Units : Pixel Shader Units : Texture mapping units : Render Output Pipelines

It is important to note, however, that these laptops are not the state-of-the-art in mobility. Embedded hardware has always had a strong foothold as the ultimate portable computing device. However, embedded hardware has not evolved to the point where it can support a modern programmable GPU. In fact, embedded hardware has just begun to adopt the PCI-express bus architecture, so there is a potential development time of years before embedded hardware has the capability to support programmable graphics hardware. Compounded onto this problem is the issue of driver compatibility with embedded hardware and common operating systems. Although Linux, Unix, and Microsoft operating systems exist for most embedded hardware platforms, the driver support is usually poor for the hardware on these embedded systems, resulting in additional development time to write driver software and potential conflicts if additional sensors are added to the computing platform.

## 3.2 Hardware Integration

After all the hardware was chosen, it had to be integrated. Timing was the major problem that had to be overcome during integration. As mentioned previously, the image-aided inertial system works by propagating IMU data and image sensor data to determine position. In addition, the trajectory of the navigating body is used to predict feature locations for an image correspondence search. The results of feature matching are used to correct the system trajectory via a feedback mechanism in the Kalman filter. Accurate timing is paramount. If the image sensor readings are not integrated into the projected IMU trajectory at the correct time, the feature correspondence predictions will be incorrect, causing incorrect trajectory corrections due to the feedback mechanism. In addition, binocular vision requires precise timing and synchronization between the image sensors. There are two common ways to approach system timing. One can either generate a master clock signal and synchronize all the hardware components to this signal, or use an existing hardware component's timing and use the timing offset for all other components.  The IMU contains internal timing functionality, so the decision was made to use existing hardware for timing.

To synchronize the cameras, a single camera was designated as the master camera, which was triggered internally by the software.  This allowed the control agents to signal image capture, which was very important for deadline timing.  Additional, “slave” cameras blocked on the hardware trigger signal, sent from the general-purpose output strobe of the master camera.  Once the hardware trigger signal was received, the slave cameras would un-block and capture an image.  Timing tests were performed by outputting the system time immediately after the capture trigger signal for both cameras. The tests showed that the slave cameras captured images at the same system time as the master camera.  However, the system time had millisecond precision, so the slave camera cannot be assumed to capture an image at exactly the same time, only within a millisecond of the master camera, which still results in very closely-matched images.

The IMU also had to synchronize timing with the cameras when an image was taken.  This timing synchronization was used to provide an indicator to the control mechanism to stop propagating IMU data.  The Crista IMU has an additional timing input, the pulse-per-second (PPS) counter that is normally connected to a GPS receiver. The strobed signal was connected to the PPS input, which served as a signal to reset the internal PPS timer on the Crista IMU.  The control agents could then stop IMU propagation when a PPS reset event was observed.  The IMU also had another internal timer which was used to provide master clock timing.  By using the internal timer for the master clock, the images could be accurately timed as an offset to the master IMU time, denoted by a reset of the IMU's PPS time. The hardware connectivity is shown in the diagram in Figure 3.1.

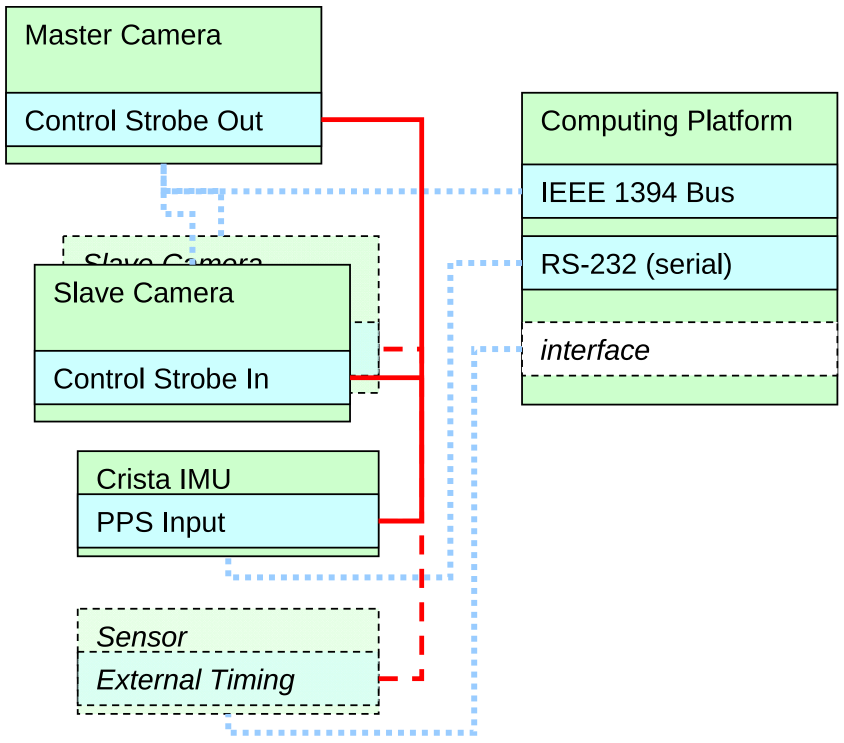


Figure 3.1: Hardware Connectivity Diagram. Control signals (bold red lines) are sent via the master camera to the slave camera(s) and the IMU, providing camera synchronization and image timing for the rest of the system. Connectivity to the computing platform (dotted blue lines) is achieved by IEEE 1394, USB and RS-232 interfaces. Additional sensors can be attached to the system as shown by the dashed line components.

The last step for hardware design was coming up with a portable power solution. The default way to power the sensors was to use a separate AC adapter for each attached sensor, as well as one for the computing platform. Simply using vision sensors and an IMU, the system had three bulky AC adapters that had to be attached to an external power supply during mobile operations.  The decision was made to come up with a more practical solution that utilized a single power source.  The power to IEEE 1394 devices can come from an external power source or from a powered bus.  This was sufficient for the image sensors, but the IMU also required power.  This was accomplished in the final system design by tapping in to a power source that came out of the general-purpose output line from the image sensors.  The power for the cameras (and the general purpose output) comes from the IEEE 1394 firewire bus.  This power source was sufficient to power two image sensors and the IMU.  It is likely that many more devices could also be powered by tapping in to this bus for future sensor platform designs.  The final sensor platform configuration is shown in Figure 3.2.

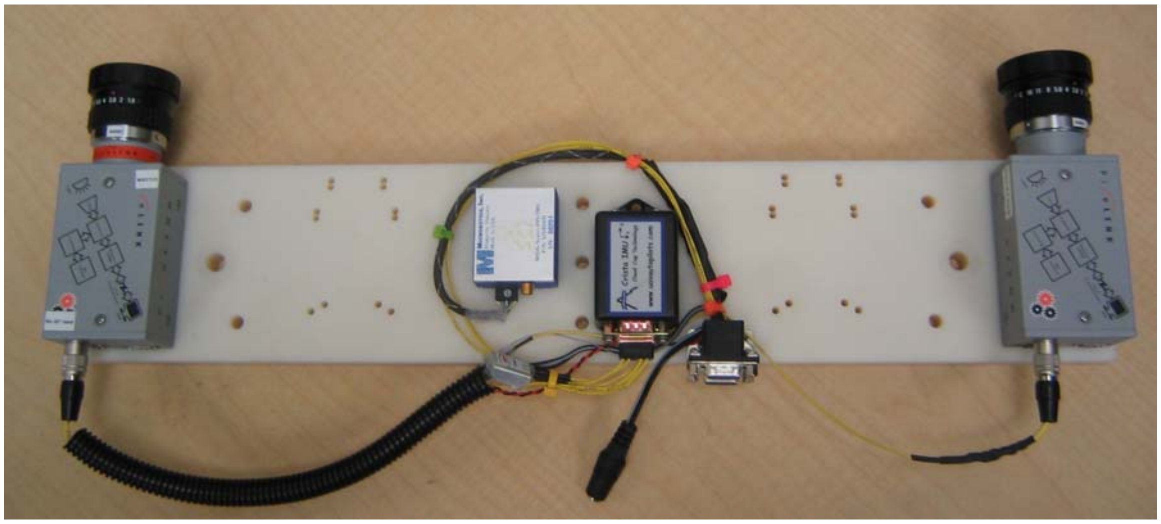


Figure 3.2: Sensor platform for the basic navigation system.  The final sensor platform uses binocular vision and a choice of two connected IMUs; the Crista IMU and the MIDG-II.  Connectivity to the cameras is provided by a daisy-chained Firewire bus.

## 3.3 Software Design

The software design for the navigation system was the most complex, risky, involved, and rewarding part of the entire thesis. As mentioned previously, the system was built as an improvement to the existing work by Major Veth. The major area that needed improvement was the image processing algorithm, since that was the biggest barrier for real-time operation. The GPGPU concept was studied, and several designs were considered. The initial design was developed using C# (pronounced C-sharp), Microsoft's OO language, and Accelerator, Microsoft Research's hardware-abstract GPGPU API, to perform GPU-accelerated image processing. During the development of this initial system, the open-source OpenVIDIA project (Fung 2005) was released. At that time, focus shifted from the C#/Accelerator project to the current solution, utilizing C++ and OpenVIDIA as the foundation for GPU-accelerated image processing.

The design of the current system was approached in the RUP manner, with weekly requirements meetings, risk mitigation, use cases, weekly releases, and UML documentation. In addition, the architecture was carefully planned and implemented to provide maximum extensibility for additional sensors and functionality without disturbing the core GPU-accelerated feature extraction system. As sensors were integrated into the solution, the need for concurrency arose, along with mutually-exclusive locks and condition objects for thread-safe access to data. The concurrency restrictions drove the need for the blackboard multi-sensor fusion architecture to help minimize the complexity and eliminate concurrency issues such as deadlock.

The current software system is comprised of four main components; the sensor agents and controllers, the feature extraction and image loading component, the navigation state and prediction component and the view and user interface component. It differs from the traditional MVC architecture because the use of GPGPU image processing led to a situation where the view component contained functionality that encompassed both the user interface (UI) and a portion of the model and controller. The model and controller roles were removed from the UI by the creation of an abstract object to help encapsulate the GPGPU functionality and decouple it from the UI. However, some controller objects still retained view component functionality due to performance constraints. All four components are joined together by a high-level composition object, the main controller. The composition pattern (Gamma 1995) is a well-known software design pattern that integrates well into component-based software architecture. Each component's design is described in a separate sub-section.

### 3.3.1 Use of the Rational Unified Process

A typical RUP iteration for this thesis began with the typical inception phase requirements meeting. A requirements meeting with the "user", Maj. Veth, evaluated and discussed new requirements for their benefit, applicability and feasibility, given time and hardware constraints. Examples of requirements were DCAM camera interoperability, multiple processing modes, network capability, and the ability to load navigation states from a configurable plain-text file. Requirements meetings typically resulted in a new additional set of use cases, or modification to existing ones. Risk mitigation was performed next, during the elaboration phase. If an addition or change required an architectural change, it was identified as high risk and performed first. If the change added functionality without any substantial change to the system, then that change was performed next. Lastly, any changes or additions that resulted in improved system stability, compatibility, or future extensibility was performed. These changes were also often performed after a release, but before a new iteration began, since the changes usually required in-depth research into the external libraries and packages, such as STL and GLUT. The actual programming during the construction phase was followed by a release.

During the release phase, the code was reviewed and edited to improve readability and understanding, comments were added, and the API was modified for better information hiding and protection. Lastly, the release was usually put on the AFIT network for access by the users. The AFIT network was initially used rather than a Concurrent Versions Systems (CVS) repository because of the limited number of users and their familiarity with the sharing medium. The system was later transitioned to a CVS repository on the AFIT network after the project took on more developers.

### 3.3.2 Hardware-abstract approach

The initial navigation system design used a hardware-abstracted language, Accelerator, along with an implementation of SIFT written in C#. C# was chosen because unlike other common object-oriented languages like Java, performance-restricting overhead could be easily removed. For instance, garbage collection and array bounds-checking in Java can be computationally burdensome. However, the standard Java specification has no way to turn it off. C# provides a simple way to turn off features like garbage collection using simple compiler directives. In addition, there was existing GPGPU API support for C# in the form of Accelerator.

The C# implementation began with a CPU-based version of SIFT obtained from (Nowozin 2004). This implementation was originally used for panorama building, using SIFT features for image correlation. However, the implementation was slow and unrefined. The API and process were cleaned up and a custom graphical user interface (GUI) was created for visualization of the SIFT features. After the SIFT implementation was modified to compile and run on both Windows and Linux operating systems, Accelerator was integrated into the implementation.

Accelerator is a GPGPU API that abstracts the details of GPU hardware out of the language. Rather than programming using textures and vertices, the Accelerator programmer works with arrays and kernels (Tarditi 2005). However, the downside to Accelerator was that the language compiles to DirectX API calls, which are transformed into GPU hardware assembly language. The extra layer of abstraction makes Accelerator a slower language by default than applications written in DirectX shading languages or GPU assembly. In addition, although Accelerator was advertised as being as efficient as hand-written GPU shaders, the application had to be written entirely for the GPU to get that benefit. Otherwise, the image data for GPU rendering had to constantly be written to and from system and video memory, which is a time-consuming process. The initial integration of Accelerator into the C#-based SIFT application resulted in a very slight increase in performance. The speedup achieved was negligible in most cases, cancelled out by the time needed for Accelerator to initialize the GPU and the time needed to transfer data back and forth between the CPU and GPU. The biggest gains were found when processing large images in rapid succession, but these gains resulted in a system that performed feature extraction at less than 1 Hz, which was slower than desired for real-time computing.

### 3.3.3 OpenGL approach

The decision to move from the C#/Accelerator-based platform to C++ was motivated by two factors; performance and OpenVIDIA. OpenVIDIA was released as open-source software at approximately the same time the C# system's initial performance data was being analyzed. It is advertised to do exactly what the navigation system needed; SIFT using the GPU. In addition, OpenVIDIA is written in C++, which has better math performance characteristics than C# or Java (Cowell-Shah 2004) and Cg, which is cross-platform capable, unlike Accelerator. Therefore, the decision was made to move the software platform to a full C++ implementation.

OpenVIDIA uses a mix of C++, OpenGL API calls, and Cg to perform SIFT-like feature extraction using the GPU. OpenVIDIA was written on an Ubuntu Linux platform. It uses Mark Kilgaard's GL Utility Toolkit (GLUT) (Kilgaard 1996) for the windowing system and the GL Extension Wrangler (GLEW) for its OpenGL API calls. Feature extraction consists of first using a Harris corner detector to find points of interest in an image that has been applied to a texture. These points of interest are placed in another texture and used as a lookup table for SIFT feature vector calculation. A 128-element SIFT feature vector is then computed and written to each row of the lookup table texture. Histogramming, necessary for orientation calculations in SIFT, is also performed on the GPU in OpenVIDIA using the cosine function from the Cg fragment shader API. This is especially impressive because histogramming normally requires a decision tree (nested if-else statements), which GPU hardware does not natively support. In addition to feature tracking, OpenVIDIA can also be used to perform a Hough transform or many other types of image filtering using the GPU.

The software platform for this thesis was built on OpenVIDIA as the foundation for the feature extraction component. To ensure compatibility with this foundation, the same GLUT, GLEW, and Cg libraries had to be included. Since cross-platform compatibility was desired, the standard template library (STL) was used for almost all C++ data structures and I/O that was not written by the author. The STL is a collection of template-based data structures and algorithms with common implementations in Unix and Windows. External libraries were only used if there was no existing solution in the STL and the external library was cross-platform capable. The basic program flow was to load an image from either an image sensor or from file, extract features for this image, then redisplay the extracted features to the user. For navigation, the program would have to run the navigation component either serially or concurrently with feature extraction, and use the extracted features for feature correspondence. Requirements meetings during the RUP process led to more use cases and a much more full-featured program, as described in the following sub-sections.

### 3.3.4 Feature Extraction Component

The feature extraction component of the system software was heavily coupled to Fung's OpenVIDIA implementation. To minimize coupling of OpenVIDIA with the rest of the system, and allow for future implementations to use a different feature extraction library, the software used interface objects to encapsulate the feature extraction routines. The Texture Manager class was created to be this interface to the OpenVIDIA library. The Texture Manager class also encapsulates the render-to-texture functionality needed for feature computation. OpenVIDIA provides the API to compute features, but does not perform feature extraction from an input texture until the render-to-texture step has been completed. Additionally, to streamline performance, the Texture Manager class was given the ability to draw features and feature-related display objects using OpenGL commands. This was a departure from the MVC design, because the feature extraction step is a function of the model or controller component, while drawing data to the screen is typically a function of the view component. This was a necessary departure because both feature extraction and drawing are GLUT window-based, and the Texture Manager class serves as a GLUT window manager.

However, the task of coordinating and loading images to the multiple Texture Manager objects was assigned to a higher-level class; the Image Loader. The Image Loader class was first designed to load static images from a file. The idea was to get the system to reliably extract features from static images, then move on to dynamic camera images. There was no file-loading capability built in to OpenVIDIA, so that capability had to be added.

Once SIFT feature extraction was working on image files, the next step was to get the software to perform SIFT feature extraction on images from attached image sensors. OpenVIDIA had a pre-existing camera control API that used a Linux DCAM library package. However, there was no existing DCAM library in Windows. In fact, most IEEE 1394 camera manufacturers create their own software development kit (SDK) for control and configuration of their cameras. These SDKs usually only work with one driver and one camera, which poses a compatibility issue if the image sensor is ever changed to a different type, or if the sensor platform contained image sensors from multiple manufacturers. To eliminate this problem, the software system was built to utilize the Carnegie-Mellon University (CMU) DCAM drivers. The CMU DCAM SDK provided an API that was common to all DCAM-compliant cameras and drivers that provided most of the DCAM-specified functionality for any attached DCAM-compliant camera. This made it equivalent to the Linux DCAM package found in OpenVIDIA. Much like how the Texture Manager class encapsulated much of the OpenVIDIA and OpenGL functionality, the DCAM functionality was encapsulated in a Camera Controller class that served as the interface to the camera API.

The next step after getting image sensor feature extraction working was to get it working on a binocular image sensor setup. As mentioned in the hardware section, the image sensors were set up so that the master camera would be triggered in software and send a hardware trigger signal to the slave camera. The hardware triggering functionality of the CMU DCAM API works by forcing the slave camera to block the calling thread and waiting for another signal to unblock it. Unfortunately, at this point the software was single-threaded, so processing could not be performed while blocking on camera input.

The problem was solved by associating each Texture Manager with a different Camera Controller object and GLUT window. Each Camera Controller was then able to run in a separate thread, and the Texture Manager objects associated with that window could then block, if needed, waiting for an external trigger signal. The other threads could continue processing as normal. This change required the ability to render to multiple windows, which GLUT did not support. The GLUT library was switched out at this point to the FreeGLUT library, which fully supported rendering to multiple targets. Multi-threading is described in more detail in the sensor control component sub-section.

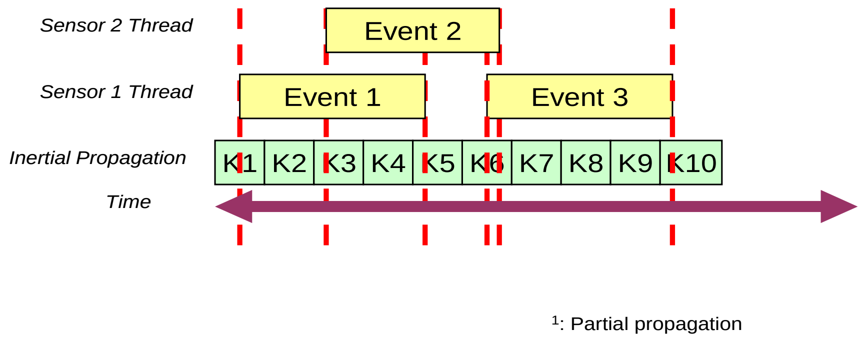
### 3.3.5 Sensor Control Component

Once feature extraction was working in real-time, the next step was to integrate additional sensors, such as the IMU. As mentioned previously, additional Camera Controller objects had to be run in separate threads because of the blocking nature of external triggering. Since other attached sensors had the potential to block while waiting for input, the sensor controllers were built so that they also ran separate from the main thread of execution. Half of the sensor controller design was interfacing with the hardware API. The sensor controller had to have an efficient routine to interpret the signals coming from the hardware and process it into usable data. The other half of the sensor design was data access. Once a sensor controller acquired data, it had to find some way to either send the data somewhere for immediate processing, or to write the data somewhere to be accessed at a later time.

The concept of multi-sensor fusion played a large role in the design decisions made for concurrent sensor access. The system was designed to maximize hardware access and minimize sensor data writing time, which could only be done by writing sensor data immediately after it was acquired. For example, if a sensor performed immediate data processing, then it could possibly have to wait on a locked or shared resource. It also might execute a task that takes a long time to complete, such as IMU propagation. The longer the wait or task, the more chance there was that the sensor controller would miss incoming data from the hardware. In dead-reckoning navigation systems, missed data results in inaccurate navigation predictions. Therefore, the decision was made to immediately write the data somewhere that could be accessed at a later time.

However, this brought about the classical problems of concurrent access to data shared among multiple threads of execution. The problem that could occur is that a thread might try to access a memory location as another thread is writing to the same location. Mutually-exclusive locks and condition objects were used to regulate access to shared data. The threading system, locks, and condition objects came from OpenThreads (OpenThreads 2007), which is a cross-platform concurrency library extension for C and C++. OpenThreads is modeled after the java concurrency package and uses a Posix-like API.

The blackboard architecture was used to manage the synchronization data structures and provide a common repository for sensor data, despite what sensors were actually present. This way any attached sensor could write to one common area, and all the sensor data could be retrieved from the same common area. Two time-ordered queues were used in the blackboard, one that contained IMU data and another that contained Event objects, which are simply sensor data wrapped by the time that the sensor acquired the data and the type and ID of the sensor controller. Event objects are an example of the decorator pattern (Gamma 1995). The two were kept separate because the IMU data drove event access, and events were often written to the event queue long after the corresponding IMU packet had been written to the IMU queue, as illustrated in Figure 3.3.



| | |
|---|---|
| K1 | K1¹ |
| K2 | Event1 |
| K3 | K1¹ |
| K4 | K2 |
| Event1 | K3¹ |
| K5 | Event2 |
| Event2 | K3¹ |
| K6 | K4 |
| K7 | K5 |
| K8 | K6¹ |
| K9 | Event3 |
| Event3 | K6¹ |
| K10 | K7 |
| | K8 |
| | K9 |
| | K10 |

Figure 3.3: Event queuing and scheduling. The boxes represent tasks being performed by the threads of execution. The left edge represents the time when the task was started, the right edge represents the time when the task was queued to the blackboard.

Because of this relationship, timing synchronization between sensors was crucial. If timing was not synchronized, then sensor data would be integrated into the IMU trajectory at the incorrect time. As IMU data was acquired, it was written to the IMU queue and the master clock time was updated in the model. As other sensors acquired data, they retrieved the current master clock time, so that time could be used as the Event time when data was ready to be written to the Event queue. As Figure 3.3 shows, if timing was not taken into account for order of execution, the task order would be in the order indicated in the left column. However, the correct order of execution, based on when the event actually occurred is very different, indicated by the order of events in the right column. Examples of Events that were put on the Event queue are Scene data (Features extracted from images using OpenVIDIA), GPS position data from a GPS receiver, and LIDAR data.

The sensor controller interface was developed to be implemented by all sensor types attached to the system. The sensor controller interface for non-image sensors in the system simply collected and enqueued data for use by control agents. Such implementations were simply controller objects that changed the model. However, each image sensor was matched to a GLUT window, which performed both feature extraction and display functions. Therefore, the image-based sensor controllers were controller objects that affected changes to the model and the view components of the MVC paradigm. Despite these differences, there was enough similarity between the imaging and non-image sensors to make them derive from the same base class.

The sensor controller interface worked in a simple threaded loop. It initialized, checked the hardware to determine if data is available, and interpreted that data as necessary until it had a data packet that can be processed. In most implementations of the sensor controller, data packet building was performed in a state machine. The data packet was then sent to the blackboard and written to the appropriate queue. Once the data was written, the thread returned to the hardware checking loop. The sensor controller interface was implemented for a Crista IMU and DCAM camera sensors, as well as a SICK scanning LIDAR and a Novatel OEM-4 GPS receiver. The sensor controller interface conceptual diagram is shown in Figure 3.4.

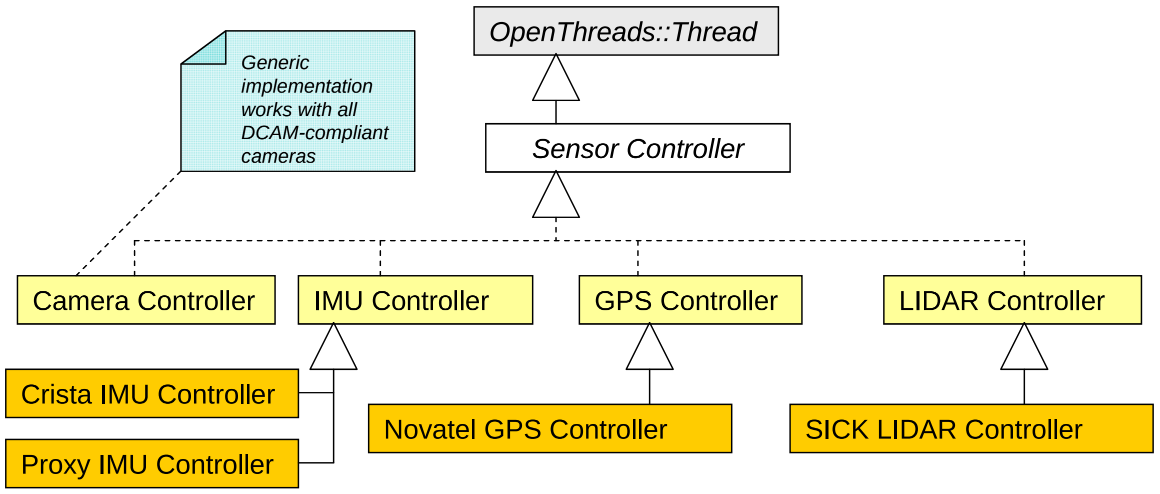


Figure 3.4: Sensor controller interface. The sensor controller interface is and abstract class derived from the OpenThreads::Thread class, and is implemented by the camera controller, IMU controller, GPS controller, and LIDAR controller classes. These concrete controller classes inherit properties from the Thread class and implement the methods in the SensorController class.

### 3.3.6 Navigation Component

Once sensors had been integrated and working concurrently with feature extraction, the data was used for navigation by creating the navigation component. The navigation component was mostly designed after the previous work by Maj. Veth. The existing system had been written in MATLAB, which is an interpreted language built on C, and does not have the power, flexibility, extensibility and compatibility of C++. The major problem with converting a program from MATLAB to C++ was the lack of built-in linear algebra functions and data structures. Matrices, vectors, eigenvalue computation and Cholesky decomposition were just a few of the functions that were native to MATLAB, but had to be duplicated in the C++ application. Much of the functionality was obtained by using, modifying, and improving the Template Numerical Toolkit (TNT), a C++-based linear algebra library from the National Institute of Science and Technology (NIST). However, there was so much functionality lacking from the TNT package that a large portion of time was spent creating, testing and debugging linear algebra functions written to emulate MATLAB functionality. In the end, the navigation component was designed as a composition of four major subcomponents; the navigation state, the navigation state machine, the Kalman filter, and the landmark tracking component.

The first subcomponent designed was the navigation state. This was a necessary foundation because every other subcomponent changed or used the navigation state. The navigation state included location information for both the navigating body and the landmarks identified in the system as well as direction cosine matrices (DCM) for sensor and platform pose information. The location information includes both an initial position, in WGS-84 coordinates, as well as a current position relative to the local navigation frame, that could be combined with the navigation state to compute the current position relative to the WGS-84 ellipsoid. The state also includes a set of the currently tracked landmarks, their location, how long they had been tracked, and the error associated with each tracked landmark. The navigation state also contains hardware parameters, such as the camera and lens focal length parameters, IMU biases, and other hardware information that was used by the navigation component. These hardware parameters make up the navigation state's hardware model.

The hardware model connects the navigation component to the corresponding hardware physically connected to the system. The camera model consists of the camera pose relative to the sensor platform, the image width and height for capture, the principal point, camera pose information and lens parameters needed to correct image distortion. The IMU model consisted of the pose of the IMU in relation to the navigating body as well as biases for the gyros and accelerometers in the IMU.

Once the navigation state and hardware model were defined, a landmark tracking sub-component was designed to detect and predict landmarks from the features extracted in images. Landmark tracking is based off of the image correspondence theory presented in Chapter 2. The landmark tracking function initially takes an image, extracts features, and chooses landmark candidates from those features. Landmarks are chosen based on the number of cameras available and the distance between features. If there is more than one camera available, the landmark detection algorithm tries to match features between both images, predicting the location in one camera's image to a corresponding location in the other camera's image, based off the camera model information. The landmark detection sub-component also gives preference to features that are more distinct and further apart from each other, which minimizes false positive identification during feature correspondence searches. These features are chosen via a weighting function applied each time a new landmark candidate was evaluated. The function gave more weight to those candidates that had a sufficiently large SIFT scale factor and a minimum

Mahalanobis distance from currently-tracked targets. The highest-weighted features are assigned as new landmarks and given a 3D coordinate and uncertainty information in addition to their basic SIFT feature information of scale, orientation and descriptor. This gives the navigation system a representation of where the landmark was in the real world, versus the feature location on a 2D image plane. The landmarks are stored in the navigation state, in a STL vector object for fast searching, sorting and efficient memory management.

The landmark detection sub-component then performs a search for tracked landmarks in the next image when it became available. The search is a constrained predictive feature search, where full feature extraction is performed on the next image, but only a subset of these features are actually matched against the previous image's tracked landmarks. The subset of features is chosen by predicting the tracked landmarks from the previous image into corresponding locations in the current image, using the IMU trajectory and the pose information from the navigating body, cameras, and IMU.

The last subcomponent designed was the Kalman filter, which provided a feedback and fusion mechanism for the system. The Kalman filter, as mentioned in Chapter 2, is a recursive estimator. It can take the previous state and use it to predict the current state. The navigation state for the navigation system presented in this thesis consisted primarily of position and pose information output by the Kalman filter. The Kalman filter is capable of true dead reckoning, where the state input and output could be solely based off of the IMU sensor data. The Kalman filter is also capable of image-aided inertial navigation predictions, where the results of feature correspondence are integrated into the dead-reckoning solution. These results are used to correct the current solution, and fed back into the Kalman filter to refine future solutions by adjusting the IMU readings. The feedback-enabled Kalman filter is illustrated in Figure 3.5.

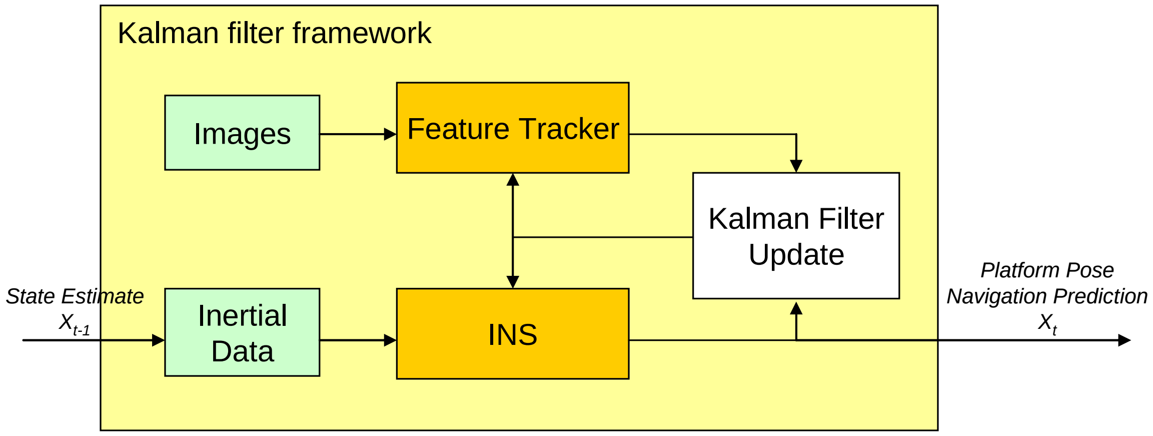


Figure 3.5: Kalman filter for image-aided inertial fusion. The Kalman filter in this diagram fuses image and inertial sensors by using the results from feature correspondence to correct the IMU trajectory. The trajectory corrections are then fed back to improve future predictions.

The last navigation subcomponent, the navigation state machine, was created as an event-driven navigation processing thread. As mentioned in Chapter 2, the concept of an image-aided inertial navigation system is to process inertial data until the time when an image was taken, use the image to correct the IMU trajectory, then repeat. However, to incorporate additional sensors that may have readings at other times, an event-based processing system was needed. At the heart of event-based processing is the Event data structure and the Event queue, described previously in the sensor control / blackboard description. The navigation system uses the time of the next Event on the Event queue to decide the next action to take. For instance, if the IMU packets on the IMU queue are all tagged with a time that is less than the next Event on the Event queue, then IMU propagation needs to occur. If the next Event is an image, then feature correspondence needs to take place. If the next Event is a GPS packet, then the GPS update routine for the Kalman filter needs to be invoked. The Event processing loop is switch-based, making it flexible and extensible for a number of different sensors and sensor types.

The navigation state machine processing takes place in a separate thread from the sensor control and feature extraction components. This is beneficial because the concurrency allows navigation predictions to take place independent of sensor controls. This way the sensor controllers were decoupled from the navigation system, and were not limited by the speed of the navigation prediction. In addition, this allowed the sensors to run in a non-navigation mode, useful for data-collection experiments. The sensor controllers merely queued Events to the blackboard, and the navigation system used these Events as they become available. However, this can also be dangerous, because the decoupled sensors could possibly acquire data faster than the navigation system could process it. In this situation, the Event queue would eventually be overrun by the sensor data. This situation can be accounted for by increasing the speed of the navigation algorithm, decreasing the sensor capture rate, or improving the speed of the computing platform via hardware upgrades.

For this navigation system, navigation state prediction is kept relatively simple to ensure that it runs at a much faster rate than sensor acquisition. In addition, the computation-intensive image processing step is coupled to the image sensor data acquisition, which limits the sensor rate so that it is much slower than the navigation prediction rate. The only problem with coupling image processing to the data acquisition step is that the image capture rate is limited by the speed of feature extraction.

### 3.3.7 View Component

The final system component integrated was the view component. This component was developed last because it was the last layer of abstraction from the model to the user. Early iterations used a simple command-line driven interface, where the user declared runtime options as command-line parameters, which were then interpreted before the program started computation. This approach was problematic because mistyped options would result in incorrect application behavior or failure. The user had to know a set of switches and commands that corresponded to various application options. In addition, the user could not switch these options while the application was operating.

The weaknesses of the command-line approach led to a more interactive UI, where the user could dynamically alter parameters as the program was running by overlaying a menu on the view screen and introducing a keystroke capture routine. However, this approach used valuable rendering resources and was still short of a professional user interface. The final design used a menu-driven UI that was rendered in a separate window and a separate thread. While this used slightly more computer resources, it removed the UI component from the feature extraction component by separating the menu windows from the image processing windows.

## 3.4 Real-time Constraints

The software system was designed with real-time constraints in mind. Real-time constraints, as mentioned in Chapter 2, consist of a deadline from event to system response. The event associated with this system was the acquisition of sensor data by the sensor controller. The system response was the navigation prediction output by the navigation component. The deadline time was affected by the capture rate of the sensor, the speed of the sensor processing routine, and the time to compute a navigation prediction from the sensor data. These variables meant the deadline would vary based on the sensor type, settings, and computing platform. In addition, a missed deadline would not result in system failure, so it could be tolerated and the previous navigation prediction could be used for an approximate location. Therefore, the system was best suited to a soft real-time system.

The blackboard architecture and concurrent sensor model offered a very flexible platform for real-time system design. Since the sensors were decoupled from the navigation computation and from each other, they could complete their task without interruption. This meant the system was non pre-emptive, and scheduling could be performed by the operating system (OS) rather than a real-time scheduler. In addition, since the OS performed scheduling, all threads had equal priority, eliminating problems of priority inversion. This design worked well with the existing computing platform, which utilized multi-processor computing architecture, since the typical real-time scheduling algorithms (LL and EDF) are sub-optimal for multi-processor computers.

The navigation system was designed as a soft real-time system with a non pre-emptive, single-priority scheduler. One goal of this system, like all other real-time systems, was to minimize the deadline. The deadline for a non pre-emptive scheduled system is the time required for the shortest task plus the time required for the longest task. The tasks for this system are divided among the system components. In terms of complexity and required computation, the shortest task in the system is sensor acquisition, while the longest task is processing a Scene Event for navigation. Therefore, minimizing the time for the sensor acquisition as well as the time for Scene Event processing will result in the lowest deadline time for this system. Since the deadline is the time between the sensor acquisition and navigation prediction, the latency of the system is equal to the deadline. The system was therefore designed to minimize the task times for sensor acquisition and feature extraction in order to minimize the real-time deadline and latency.

## 3.5 Summary

This section provided an overview of the methodology used to design and implement the software and hardware used for this navigation system. The details of hardware design and integration were discussed, focusing specifically on compatibility and timing. Software design was approached in a RUP manner, with the riskiest components being developed first. In addition, weekly prototypes of the software were developed, and functionality was added to each prototype release. This meant that architecture issues were handled early on in development and functionality was added so that it would not disrupt the rest of the existing software.  Software design was broken up into four components, each representing a major part of the software.

# IV. Analysis and Results

The first goal of the system was to improve upon the previous system by performing accurate, precise navigation in real-time, rather than post-processing navigation data. To ensure accuracy, the system needed to capture and process image data at a rate that would correct errors in the inertial measurement data. The previous work in this area found that post-processing images captured at 2.5 FPS would stabilize the feature correspondence search using consumer-grade inertial sensors. Therefore, the new system had to capture and process navigation data at a rate that exceeded 2.5 Hz. The second goal was to minimize the latency, or the amount of time between when the sensor data was acquired and when the navigation prediction based on that sensor data was computed. This required first knowing what the system latency was, then overcoming any bottlenecks in sensor acquisition and navigation prediction computation to minimize the latency.

## 4.1 Test Plan and Setup

In RUP development, projects must be tested according to a test plan, which outlines the tests to be performed their timeline. The test plan for this system was driven by development needs, which is typical of a RUP-style project, where tests are tailored to fit the needs of the iteration in which they are developed. These tests are also designed so that they are general and repeatable for future iterations. The tests developed during early iterations were component tests and integration tests. The development of a formal test model enabled automated testing using a test suite similar to Junit tests (Link 2003) during later iterations. Lastly, tests of the fully integrated system in the final iterations were performed via simulations and online tests.

### 4.1.1 Component and Integration Testing

Component and integration tests were performed when changing or adding to the system software or hardware architecture. Each component was individually tested for functionality then integrated into the rest of the system. The tests started from the user interface component and moved gradually to the more complex parts of the system that were dependent on previously tested components. The first component tested was always the view and visualization component. This meant that the windowing system, UI, and image display all had to work properly. Secondly, feature extraction was tested to ensure that any changes did not negatively affect the OpenGL or GLEW subsystem, resulting in slower or degraded feature extraction capability. The sensor controllers were tested next, since changes to the event-processing loop would oftentimes affect the rate at which sensor data was accessed from the data queues. This would expose any concurrency or data structure problems previously masked by correct operation of the event processing loop. Lastly, the navigation component would be integrated and tested, since it was dependent on the UI, feature extraction, and sensor controller components.

Integration testing revealed system bottlenecks. The first bottleneck found in the system was image access time, found when integrating the image sensor component. The OpenVIDIA feature extraction component could perform feature extractions on a 512x512 image loaded from a static file at over 20 Hz on the computation platform with the enhanced GPU. It was found that when the image sensor component was added, frame rates would drop below 20 Hz, on average. The immediate reaction was that the problem was a result of poor component design. In an effort to test the design, the static image file was reloaded every frame, mimicking the frame-by-frame image capture of the cameras. This comparison test resulted in a significant reduction in the frame rate of the static image feature extraction. Therefore the frame rate reduction was an image access time issue not necessarily a design issue. The image access time was affected by a number of different factors. File-based image loading time was a result of the speed of GPU video memory and main system memory, as well as the hard disk drive speed. Camera-based image loading time is limited by the GPU video memory, main system memory, the image sensor hardware, and the drivers used to access that hardware. To maximize the image processing rate of this system, fast camera and computer hardware components are needed.

### 4.1.2 Model-based Automated Testing

A program is a combination of algorithms and data structures [Wirth 1976]. The previous MATLAB-based navigation system and the new C++/GPU-based navigation system used very similar navigation algorithms, but were not identical, due to the differences in feature extraction, programming language, data structures, and external libraries. However, the navigation results from the old system should be used as a model for comparison of navigation predictions to the new system, given the same sensor data as input.

The model was first tested by comparing results for the same system inputs at the function-level. Given the same initial conditions, image data, IMU data and integration time, the new navigation system should predict the same navigation position as the MATLAB system. To go further in detail, the Kalman filter in the new system should output the same $X_t$ and $P_t$ as the MATLAB system. This comparison can be traced back, step-by-step, to the output of each function call for the Kalman filter and navigation system. By comparing the inputs and outputs for each function call, any discrepancies could be found and either accounted for, or fixed.

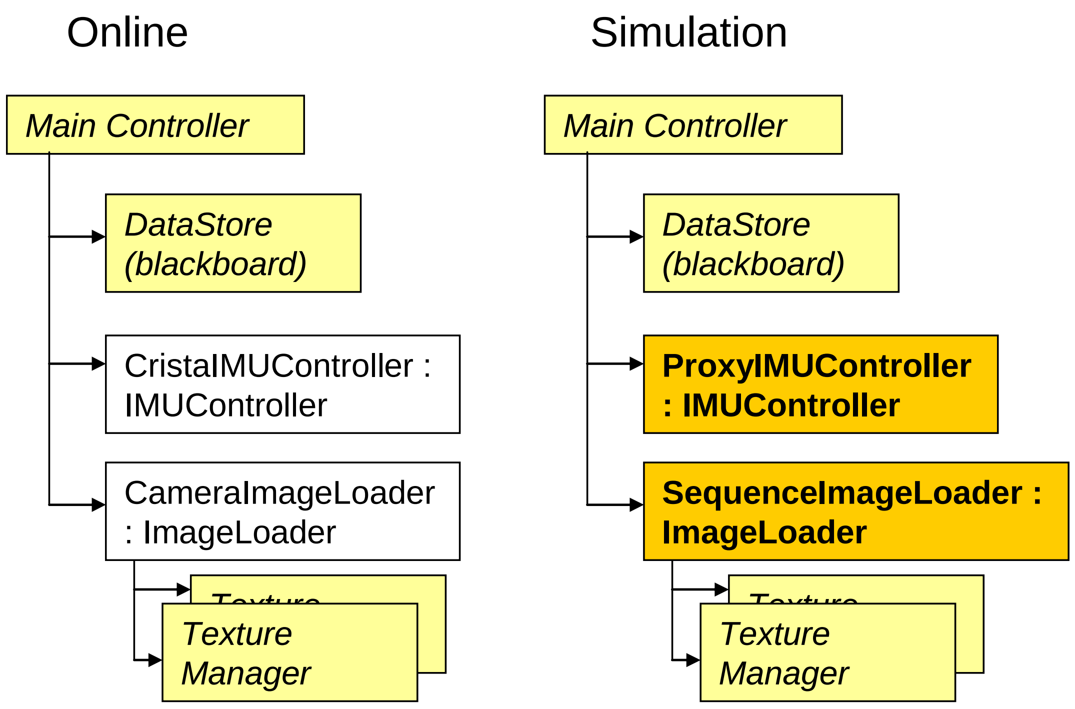
Since the comparison was at such a fine level of granularity, the tests were able to be automated at the method-level. Given a fixed input, a method was expected to output a certain value. If the output deviated from the expectation, then the method did not execute correctly. The entire Kalman filter functionality was given an automated suite of tests and the program was given a way to run in a "debug" mode to test the methods individually using a runtime option, without needing to recompile. The automated suite functionality was based on prior experience with Junit tests. An automated test first loaded the initial conditions consisting of the navigation state and any sensor inputs. Then the test input these conditions to the function, and compared the output to the MATLAB answer that was hard-coded into the test. If the values were within a tolerable range of precision, then the test passed. If the values are outside of this tolerable range, the error statistics were output to the user and the test failed. These tests were built so that they could feed into each other to create more complex tests that were incrementally verified.

The advantage of automated testing is twofold. First, functions could be verified at any point, just by inputting a different initial condition. This meant that if there was a certain point in the program that needed to be tested, the program did not need to be run up until that point, saving the programmer valuable time. Secondly, the test was repeatable, since the variables and output were all the same, the test could be run as many times as was necessary and the same output was expected. If the output changed, then the controls were set up incorrectly for the function test.

### 4.1.3 Simulations

The navigation system presented in this paper was created to process sensor data and navigation predictions concurrently. However, the availability of a model and automated tests provided an opportunity for the two systems to be compared by using data loaded from file. By simulating the presence of sensors, using sensor proxies that load the data from file, the entire program could be run in a simulation mode, useful for future debugging and post-processing efforts.

The proxy sensor controller design was aided by application of the proxy pattern (Gamma 1995; Freeman 2004). Proxy objects emulate the behavior of other classes, to simulate the presence of objects of the other class in the system. When running the navigation system in simulation mode, the system sensor data was loaded from files, eliminating the need for data coming from system sensors that were actually attached to the system. Rather than eliminate the sensors altogether and read straight from file, the sensor behavior was emulated in the proxy sensor controllers. Proxy sensors behaved in the same manner as the actual sensor controllers, except that the proxy sensor readings came from file-based sensor readings captured during data collection. This way any issues with sensor concurrency, interaction, or latency would be duplicated in the simulation mode, rather than eliminated.  This unified the program behavior across both simulation and active running mode, eliminating redundancy in testing and aiding system debugging.  The proxy-based system is illustrated in relation to the active "online" system in Figure 4.1.



Online
Simulation
Main Controller
DataStore (blackboard)
CristalIMUController : IMUController
CameraImageLoader : ImageLoader
Texture Manager
Main Controller
DataStore (blackboard)
ProxyIMUController : IMUController
SequenceImageLoader : ImageLoader
Texture Manager

Figure 4.1: System setup for simulation and online processing.  The italicized (yellow) classes are common throughout every mode.  The bold (gold) classes are applications of the proxy pattern.

Simulations began with data collection.  The navigation system was run in a data collection configuration where it saved sensor data to file.  The sensor data encompassed readings from everything attached to the system, such as image data in the form of portable grey-map files, IMU data in the form of accelerometer, gyro, and timing data, as well as optional GPS data and LIDAR data. This data was saved in a separate directory, and the proxy image loader class was configured with the correct parameters to read the sensor data from this data directory.

Once the data collection was complete, the navigation system needed a correct initial state. This was accomplished by initializing the Kalman filter and navigation state to default values, as well as determining the correct earth-centered, earth-fixed (ECEF) position for the sensor platform. Once the inputs were correctly set up, the system could be run in script mode, which worked the same as online processing mode, but loaded all sensor data from file to the proxy sensor controllers. The navigation predictions at any point in the simulation could be compared to the navigation predictions from the MATLAB model, running on the same file-loaded data. In addition, either simulation could be paused to visually compare the landmarks chosen and progress on the map.

The first simulation performed was on data obtained on a run through the hallways of AFIT in August 2006. The simulation consisted of 700 seconds of real-time sensor data. The image sensors captured images at 2.5 FPS, while the IMU sensor captured data at 100 Hz. This meant that there were 40 IMU updates propagated and integrated into a navigation prediction between each update by the image sensors. The MATLAB model completed processing the data in 2251 seconds, which was 3 times as long as the real-time operation. The MATLAB final position prediction was within 2 meters of the actual final position. The new system's performance was measured against the MATLAB model in two categories; overall speedup and prediction precision. The new system performed admirably in speedup, completing the script at an average of 4

FPS, in 455 seconds. This was a 4.94x speedup over the MATLAB solution and 1.54x speedup over real-time. The new system also performed very well over the range of data, deviating less than 1 meter from the model predictions over 450 seconds of real-time operation, as shown in Figure 4.2. However, the new solution did show a weakness when there were no selectable landmarks, resulting in poor navigation predictions after the 450 second time mark, as shown in Figure 4.3.

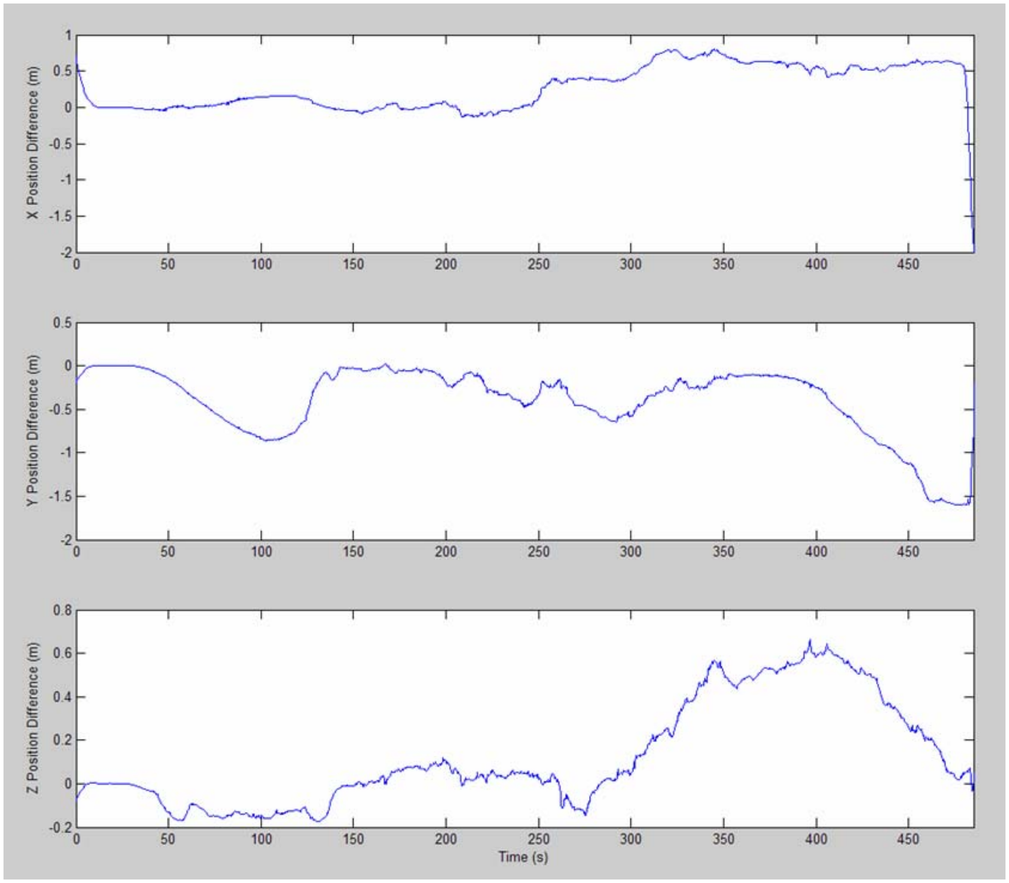


Figure 4.2: Navigation prediction comparisons during a simulation run (partial). The MATLAB model is used as the truth model, or basis for comparison, for the new navigation system. The new system deviates less than 1m from the truth model less for the time period shown.

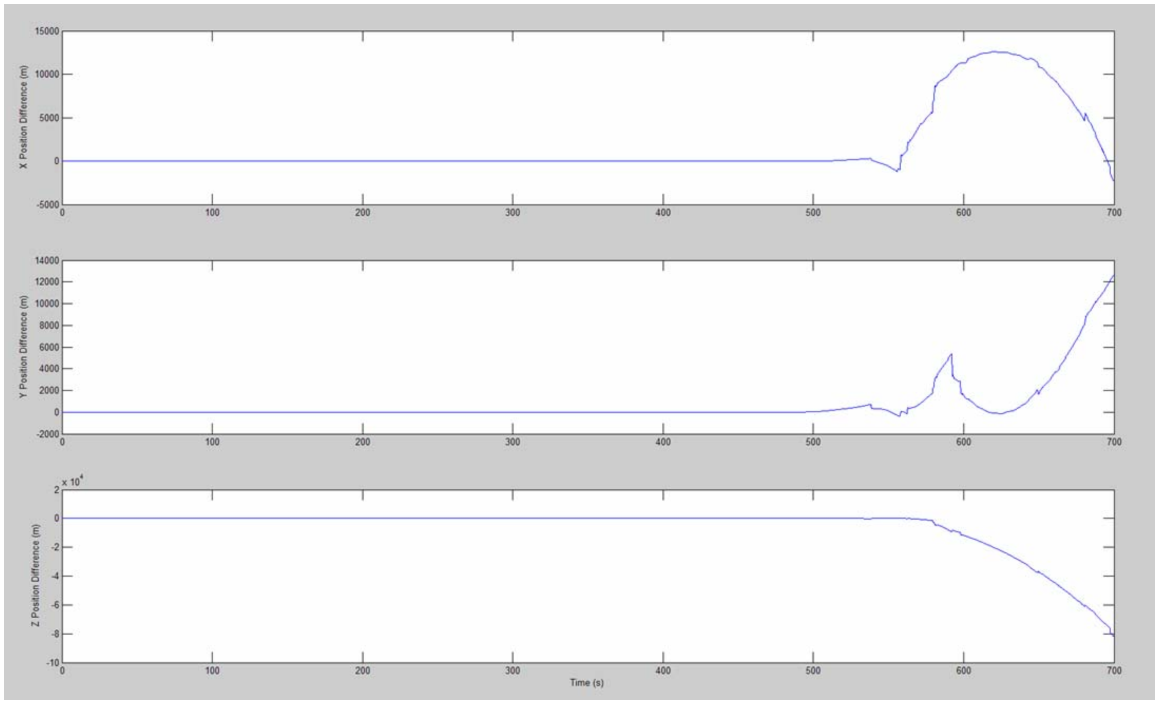


Figure 4.3: Navigation prediction comparisons during a simulation run (full). The MATLAB model is used as the truth model, or basis for comparison, for the new navigation system. The new system deviates greatly from the truth model during the last 250 seconds of operation.

The simulation results were obtained when reading features from a pre-computed file. However, the system is capable of computing features using OpenVIDIA as well. The performance data from using OpenVIDIA for feature extraction suggests that the algorithm does not compute SIFT descriptors properly. After much analysis and investigation, it was determined that the OpenVIDIA algorithm does not perform scale decomposition of images, resulting in features descriptors that are invariant to rotation, but vary with changes in scale. Although this isn't a problem for a stationary platform, it is very detrimental to a moving platform, where landmarks will vary in scale as they move toward or away from the image sensors.

## 4.2 Speedup Due to GPU Acceleration

The feature extraction component was tested by using the built-in timing functionality of the GLUT library that could determine elapsed time with millisecond precision. The time just before feature extraction was subtracted from the time just after the operation completed to determine the feature extraction time. The feature extraction component of the new navigation system was timed against the feature extraction times for the same images using an implementation of SIFT that solely used the CPU to perform feature extraction. Although this implementation used optimized image processing routines found in the OpenCV library (Intel 2007), Figure 4.4 shows that even optimized CPU-based routines perform poorly in comparison to GPU-based feature extraction. In fact, the speedup ranged anywhere from 13x to 103x speedup for a single computing platform, with the larger images resulting in a greater speedup.

There are many possible explanations for these findings. The most obvious one is that the hypothesis of this thesis is true, and GPU-based image processing is decidedly better than CPU-based image processing, even when applied in a non-conventional manner. Another explanation is that the OpenVIDIA library isn't performing a full SIFT feature extraction algorithm, and that the skipped steps are causing the difference in processing time. This may account for some of the speedup, but other data from (Heymann 2007) supports the first conclusion.

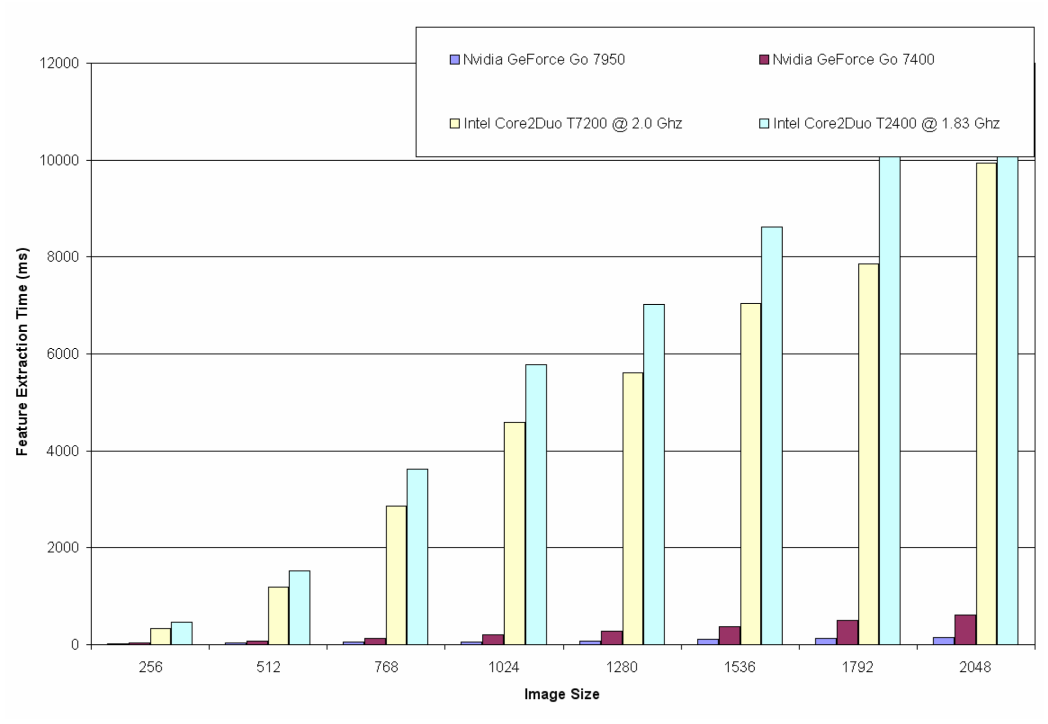


Figure 4.4: GPU vs. CPU feature extraction times. The GPU feature extraction times are between 13 and 103 times shorter than the equivalent CPU feature extraction times.

In addition to the speedup from feature extraction, offloading image processing to the GPU freed up CPU resources that could then be dedicated to other processes. To test this effect, the application was run using the gDebugger tool (Graphic Remedy 2007), which measures CPU and GPU performance for OpenGL applications. The resulting statistics were 50% CPU utilization and 20% GPU utilization, shown in Figure 4.5. Several conclusions can be drawn from these statistics. First, the GPU is underutilized in comparison to the CPU workload, so more computation should be offloaded to the GPU to better balance the workload of both computation units. Secondly, the conclusion can be drawn that the CPU is not fully utilized, even though the navigation predictions are being run at full speed. Therefore, the system could run additional background tasks, such as creation and maintenance of a feature database, or network communications, during navigation processing. Lastly, despite the complexity of image processing needed for SIFT feature extraction, the GPU is still not fully taxed. This means that this system could probably be run on slower, less expensive hardware.

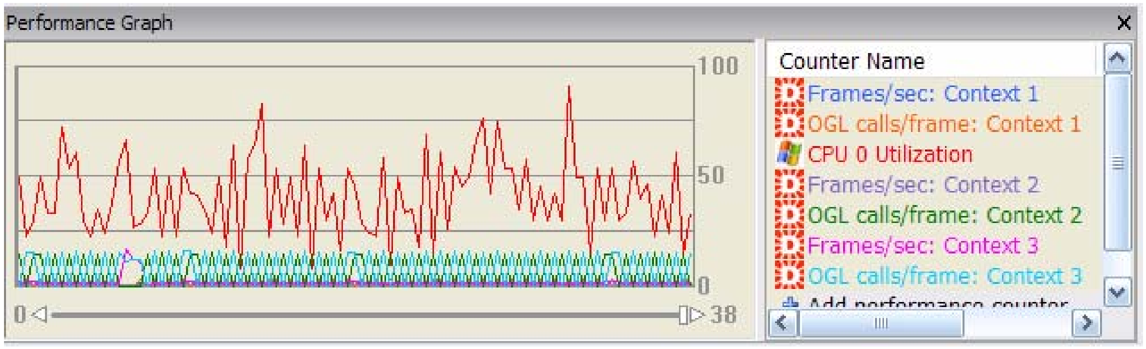


Figure 4.5: GPU vs. CPU utilization. Average 50% CPU utilization and 20% GPU utilization during normal navigation prediction operation.

## 4.3 Effects of Concurrency and Sensor Decoupling

Concurrency in this navigation system was developed out of necessity because the image sensors' blocking behavior required multithreading. The resulting blackboard system architecture not only worked well for the multi-sensor platform, it helped to minimize latency for the real-time aspect of the system. A beneficial side effect of concurrency was that the system was sped up compared to the MATLAB implementation even without the use of the GPU.

As mentioned in Chapter 3, the system was designed as a soft real-time non pre-emptive real-time system. This meant that the latency was equal to the deadline, which is a function of both the sensor acquisition rate and feature extraction time. The system was run in both simulation and online processing mode to determine what the actual latency was versus what was expected. The expected latency for a system can be estimated by adding the expected sensor acquisition time based on camera frame rate to the expected feature extraction time. For an 800 x 600 image running at 20 FPS, for example, one can estimate the latency to be 50 ms for sensor acquisition and 100 ms for feature extraction, for a total of 150 ms latency. The actual latency was measured by outputting the time that a navigation prediction was computed and the time of the Event associated with that navigation prediction. The latency varied between 80 and 300 ms based on the rate and size of image capture. Based on the results of feature extraction speedup for the system, it is apparent that the latency will also vary based on the speed of the computing platform. The latency does not vary based on the attached sensors, as one might expect. Since the sensors are decoupled from navigation predictions, sensor data acquisition times do not factor in to the deadline timing. The only sensor acquisition times that matter are the image sensors, since the feature extraction task takes the longest to integrate into a navigation prediction.

In addition to minimizing latency, the entire solution was sped up in comparison to the MATLAB solution. The new navigation system exhibited a 4.94x speedup when performing navigation predictions in post-processing (simulation) mode, without GPU feature extraction. This speedup was due to the change in programming platform, data structures, and software design of the new system.

## 4.4 Summary

This section centered on system testing and results from those tests. The test plan was developed for each iteration with a focus on testing what was needed, using tests that were repeatable at any iteration. Component and integration tests were used for debugging during additions or changes to the system and also revealed bottlenecks in the navigation system. The existing solution written in MATLAB was used as a truth model to perform function-based testing, which then led to automated testing and simulation testing. The simulation tests revealed that the new system's predictions were very close in precision to the model, and the simulation ran much faster than the MATLAB solution, faster even than the sensors could capture data for the simulation. This speedup was due to sensor decoupling, concurrency, and good software design. Lastly, the GPU acceleration for this system was tested and found to be faster than any current solution, and much faster than what was expected.

# V. Conclusions and Recommendations

The following chapter provides a summary of the research presented in this thesis. The conclusions are first presented, including what was expected, what was achieved, as well as the reasons for the conclusions that were reached. The research is put into perspective for its significance to the scientific, academic, and military communities. Several recommendations for action and future research are presented, based on the results achieved and the potential for future research. The future research was chosen from areas that have the most potential positive impact on autonomous precision navigation and map-building using commodity hardware.

## 5.1 Conclusions of Research

In the Mythical Man Month (Brooks 1995), Frederick Brooks asserts that the schedule of a software engineering project is most often the single most overwhelming factor in a project gone awry. He attributes this to the fact that humans are optimistic at estimating the scope of a project, often underestimating the amount of work that needs to be done and confusing the effort put forth with actual progress. Much like his essay on “No silver bullet” (Brooks 1995:17), Brooks’ predictions hold correct. The software system presented in this research has much outstanding potential, but it still needs more testing and debugging before use on a live system. The system has already exceeded expected speedup results for feature extraction, based on the results from previous implementations of SIFT on the GPU (Sinha 2006; Heymann 2007). In addition, the software architecture and framework for this navigation system have been built using software engineering best principles, focusing on concurrent, real-time sensor processing and navigation predictions. This has resulted in a speedup over the previous system developed at AFIT, due to software engineering improvements alone.

The navigation algorithm was built upon a proven, working system that was only capable of post-processing navigation data. The Air Force and various other industries require a navigation system that can operate in real-time during times of GPS unavailability. The system also needs to be mobile, sustainable, and operate in a variety of unknown environments. The navigation system presented in this thesis can do all of the required tasks, and perform navigation predictions that border on the accuracy of GPS. In addition, since much of the computation is offloaded to a GPU, the CPU is freed up to run other tasks, like Artificial Intelligence control systems for autonomous platforms.

## 5.2 Significance of Research

The research presented in this thesis is significant for several reasons. First, the image-aided inertial navigation algorithm is a state-of-the-art GPS-alternative navigation system. The algorithm employed by the system has been proven to work in unknown environments, while most other vision-based navigation systems require *a priori* information about their environment (Panzieri 2003; Thrun 2000; Trawny 2007). This makes the navigation system especially useful for military applications, since the military often operates in austere environments, over unknown terrain, without GPS capability.

Second, this is one of the first navigation systems to use GPGPU concepts to perform image processing. Computation can be offloaded from the CPU to the GPU, freeing CPU resources to be devoted to other tasks. Unlike an integrated circuit or

FPGA, the GPU can be reprogrammed during operation via the programmable rendering pipeline. Additionally, the GPU architecture is built for raw computation resulting in superior computation performance in comparison to the fastest commercial processors. The GPU is also commodity hardware, so it can be added to an existing system as a relatively inexpensive, disposable, GPS-alternative navigation aid. This is significant to the military and civilian community since the system can be integrated into existing platforms such as vehicles, UAVs, and munitions at a low cost and lower risk than embedded systems.

Third, the research presented in this thesis is significant because it is a compact, extensible multi-sensor fusion framework that can be used for a variety of applications. The framework has the ability to add additional sensors, such as image sensors, GPS, or LIDAR, without any major changes to the navigation algorithm. In addition, this framework is not limited to real-time active navigation processing. It can also be used for data collection and simulations that run at speeds much faster than traditional CPU-based simulation programs, due to multithreading and GPU speedup. The framework is modular, component-based, and easily expandable. It is based on the MVC architecture, so the UI components, control mechanisms, and model are mostly decoupled. This allows for a simple plug-and-play functionality, where components of the software system can be replaced with a different implementation without an adverse effect on the rest of the system.

Lastly, the navigation system was built with compatibility in mind. The system can work with a wide variety of image sensors, IMUs, and computing platforms. This makes the system truly valuable from a research standpoint, because it is portable and robust to hardware changes.  Much research performed in the academic community is dependent upon a specific platform or proprietary hardware and software.  The navigation system presented in this thesis moves away from this idiom and into a cross-platform, component-based hardware and software architecture, useful to individuals outside the academic community.

## 5.3 Recommendations for Future Research

The navigation system presented in this thesis has many areas of potential improvement and further exploration.  These areas present improvements in both performance and functionality.

First, the navigation system presented in this thesis computes the navigating platform’s position, but does not keep a database of landmarks.  However, constructing a landmark database presents a potential use as a SLAM solution or for map-building. Landmarks in the navigation system consist of a unique SIFT identification key, an error covariance matrix, orientation and scale information, as well as a 3D coordinate in the WGS-84 standard.  The landmarks can be stored and plotted to create a rudimentary map of the environment.  If additional map data is needed, additional sensors such as a LIDAR can be used to augment the database.

Second, many individuals in the navigation community (Vaughan 2002; Turker 2003) have extolled the use of multiple cooperating agents for task parallelization.  This navigation system would be very useful for a cooperative, multi-agent system, since landmark observations by other agents could be treated as an additional sensor input.

The additional sensor input would provide a more robust and more accurate navigation prediction. In addition, map-building or SLAM solutions could be performed even faster by dividing the task among several agents, as seen in (Roumeliotis 2000).

An alternative to having multiple navigation systems on multiple agents is to use a central processing station and networked sensors. By moving the computation component from the sensor platform to a central location, the sensors can be smaller, faster, and agents overall would be less expensive. However, issues of network security and intrusion detection for compromised sensor platforms would need to be addressed. This design would then be very similar to the control component of GPS, which is addressing the same issues in current modernization efforts.

The navigation system presented in this thesis greatly outperforms its predecessor and contemporaries in terms of computation time. However, there is still room for improvement. The GPU is still under-utilized in comparison to CPU usage, as shown in the Chapter 4. The major source of CPU usage was the mathematical computations for the Kalman filter during IMU propagation. There are several advantages in moving the Kalman filter calculations to the GPU. First, the entire system could possibly be sped up because of the GPU-accelerated parallel-processing capability. Second, the system would have more CPU resources to devote to other tasks, such as integration of more sensors, advanced control techniques, or even to run additional applications such as artificial intelligence. Lastly, more landmarks could be tracked if the Kalman Filter data structure was moved to the GPU. The Kalman filter data structure is a 2-D matrix that grows exponentially as additional landmark tracks and additional sensor states are added.

Therefore, a navigation platform that had more sensors or tracked more landmarks would typically take exponentially longer to compute navigation predictions. However, since the GPU is a stream processor, the time to perform calculations would only increase linearly in proportion to the matrix size, as shown in (Thompson 2002).

A final area of improvement would be to take advantage of the latest GPGPU technology by developing the navigation system using CUDA. CUDA abstracts the GPU hardware from the language API, so the programmer can program with familiar data structures and general-computing concepts, rather than using textures, shader programs and other hardware-based rendering concepts. In addition, the CUDA architecture and API supports multi-threaded environments, so the multi-sensor fusion concept used in the navigation system presented in this thesis could be maintained. Since CUDA is so new, and is geared toward the GPGPU community, it would also be worthwhile to seek out corporate sponsorship for this project.

## Bibliography

Aarts, Emile, Jan Korst, and Peter van Laarhoven. *Simulated annealing. Local Search in Combinatorial Optimization*. Wiley and Sons, Ltd, 1997.

Akl, Selim. *Superlinear Performance in Real-Time Parallel Computation*. Technical Report No. 2001-443, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 2001.

Aloimonos, John. *Visual Navigation: From Biological Systems to Unmanned Ground Vehicles*. Lawrence Erlbaum Associates, 1997.

Arya, Sunil, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. "An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions." *Proc of the Fifth Annual ACM-SIAM symposium on discrete algorithms,* 1994.

Beis, Jeffrey S., and David G. Lowe. *Shape Indexing Using Approximate Nearest-Neighbor Search in High-Dimensional Spaces*. University of British Columbia Department of Computer Science, 1997.

Bellini, C., Stefano Panzieri, and Federica Pascucci. "A Real-time architecture for low-cost vision-based robots navigation." *World IFAC,* 2002.

Bertozzi, Massimo, and Alberto Broggi. *Vision-based Vehicle Guidance*. IEEE Computer, 30(7):49-55, July 1997

Bertozzi, Massimo, Alberto Broggi, Alessandra Fascioli, and Stefano Tommesani. "Addressing Real-Time Requirements of Automatic Vehicle Guidance with MMX Technology." *In Proceedings 4th International Workshop on Embedded HPC Systems and Applications (EHPC'99) - Second Merged Symposium* IPPS/SPDP 1999, pages 1407-1417. Springer-Verlag LNCS, April 12-16 1999.

Bertozzi, Massimo, Alberto Broggi, and Alessandra Fascioli. *VisLab and the Evolution of Vision-Based UGVs*. IEEE Computer, 39(12):31-38, December 2006, ISSN: 0018-9162,

Bollella, Gregory, and Kevin Jeffay. "Support for Real-Time Computing Within General-Purpose Operating Systems." *Proceedings of the IEEE Real-Time Technology and Applications Symposium,* Chicago, IL, pp. 4-14, May 1995.

Bonabeau, Eric, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: from Natural to Artificial Systems*. Santa Fe Institute, Oxford University Press, 1999.

Bowditch, Nathaniel. *The American Practical Navigator, an Epitome of Navigation*. National Imagery and Mapping Association, 1995.

Boyle, Alan. *GPS Satellite Network Goes to War*. MSNBC.com report, March 19, 2003. http://www.msnbc.msn.com/id/3078694/.

Broggi, Alberto, Gianni Conte, Francesco Gregoretti, Roberto Passerone, Claudio Sansoè, and Leonardo M. Reyneri. "Massively Parallel Hardware Support for the MOB-LAB.*" In Proceedings AATTE - Fourth International Conference on Applications of Advanced Technologies in Transportation Engineering*, Capri, Italy, June 27-30 1995.

Broggi, Alberto, Stefano Cattani, Pier Paolo Porta, and Paolo Zani. "A Laserscanner-Vision Fusion System Implemented on the TerraMax Autonomous Vehicle." *In Proc. IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pages 111-116, Beijing, China, October 2006.

Broggi, Alberto, and Pietro Cerri. *A radar driven fusion with vision for vehicle detection*. PReVENT Fusion Forum e-Journal, 1:17-18, September 2006,

Brooks, Frederick. *The Mythical Man Month. Essays on Software Engineering*. Anniversary Edition. Addison-Wesley Publishing, 1995.

Buck, Ian, Tim Foley, Daniel Horn, Jeremy Sugerman, and Pat Hanrahan. "Brook for GPUs: Stream Computing on graphics hardware." *In Proceedings of SIGGRAPH*, 2004.

Burschka, Darius, and Gregory Hager. "V-GPS(SLAM): Vision-Based Inertial System for Mobile Robots." *Proc of the 2004 IEEE International Conference on Robotics and Automation*. April, 2004.

Cartwright, B.A., and Thomas Collett. *Landmark Learning in Bees*. Journal of Comparative Physiology, no 151, 1983.

Chen, Zhiyu, Yan Gao, and Zhizhao Liu. *Evaluation of solar radio bursts' effect on GPS receiver signal tracking within International GPS Service network.* Radio Science, 40, RS3012, 2005.

Chen, Zhenhe, Jagath Samarabandu, and Rango Rodrigo. *Recent Advances in Simultaneous Localization and Map-building Using Computer Vision*. University of Western Ontario Publishing, May 2007.

Chong, Kok Seng, and Lindsay Kleeman. *Mobile Robot Map Building from an Advanced Sonar Array and Accurate Odometry*. Technical Report MECSE-1996-10, Melbourne Monash University, Department of Electrical and Computer Systems Engineering, 1996.

Cowell-Shah, Christopher. *Nine Language Performance Round-up: Benchmarking Math & File I/O*. OSnews.com, January, 2004.

Dally, William, Patrick Hanrahan, Mattan Erez, Timothy Knight, François Labonté, Jung-Ho Ahn, Nuwan Jayasena, Ujval J. Kapasi, Abhishek Das, Jayanth Gummaraju, and Ian Buck. "Merrimac: Supercomputing with Streams." *In the Proceedings of the SC'03 Conference*, November 2003, Phoenix, Arizona.

Dana, Peter H. *Global Positioning System: an Overview*. http://www.colorado.edu/geography/gcraft/notes/gps/gps.html. Page last accessed 24 May 2007.

Defense Advanced Research Projects Agency (DARPA). *DARPA Grand Challenge Overview*. http://www.darpa.mil/grandchallenge/overview.asp. Accessed April 15, 2007.

Davison, Andrew. "Real-time simultaneous localisation and mapping with a single camera." *In Proceedings of the ninth international Conference on Computer Vision ICCV'03,* Nice, France, October 2003.

Diosi, Albert, and Lindsay Kleeman. "Advanced Sonar and Laser Range Finder Fusion for Simultaneous Localization and Mapping." *Proceedings of 2004 IEEE/RSJ nternational Conference on Intelligent Robots and Systems*, Sendai, Japan, 2004.

Dissanayake, Gamini, Paul Newman, Steve Clark, Hugh Durrant-Whyte and Michael Csorba. *A Solution to the Simultaneous Localisation and Map Building (SLAM) Problem.* Transactions of Robotics and Automation, 2001.

Dong, Jing, Shanguo Chen, and Jun-Jang Jeng. "Event-based Blackboard Architecture for Multi-Agent Systems." *The Proceedings of the IEEE International Conference on Information Technology Coding and Computing (ITCC)*, pages 379-384, USA, April 2005.

Dongarra, Jack, Piotr Luszczek, and Antoine Petitet. *The LINPACK Benchmark: Past, Present, and Future*. University of Tennessee, Computer Science Technical Report Number CS - 89 – 85, 2001.

Draper, Bruce. *Camera Models: Projection and Lens*. CS510 Lecture 2, January 16, 2002.

Foster, Kenneth, and Michael Repacholi. *Environmental Impacts of Electromagnetic Fields From Major Electrical Technologies*. EMF Project report, 2005.

Freeman, Eric, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Head First Design Patterns*. O'Reilly Media, 2004.

Fung, James, Steve Mann, and Chris Aimone. *OpenVIDIA: Parallel GPU Computer Vision*. In MULTIMEDIA, pp 849-852. ACM Press, 2005.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Garlan, David, and Mary Shaw. *An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering, vol I*. World Scientific Publishing Company, New Jersey, 1993.

Georgiadou, Yola, and Alfred Kleusberg. *On carrier signal multipath effects in relative GPS positioning*. Manuscripta Geodaetica. Vol 13: 172-179, 1988.

Girija, Gopalarathnam, JR Raol, Appavu Raj, and Sudesh Kashyap. *Tracking filter and multi-sensor data fusion*. Sadhana, Vol 25, part 2. April, 2000.

Gordon, Irnya, and David Lowe. “Scene Modeling, Recognition and Tracking with Invariant Image Features.” *International Symposium on Mixed and Augmented Reality (ISMAR)*, Arlington, VA, Nov. 2004.

GPGPU.org. *General-Purpose Computation Using Graphics Hardware*. www.gpgpu.org. Last accessed 15 May 2007.

Graphic Remedy. *gDebugger - OpenGL and OpenGL ES Debugger and Profiler*. http://www.gremedy.com/. Last accessed 23 May 2007.

Griffin, Donald. *Bat Sonar*. Time Magazine. May, 1950.

Guivant, Jose, and Eduardo Nebot. *Optimization of the simultaneous localization and mapbuilding algorithm for real-time implementation*. IEEE Trans. Robotics and Automation, 17(3):242--257, 2001.

Gummaraju, Jayanth, and Mendel Rosenblum. "Stream Programming on General-Purpose Processors." *In Proceedings of the 38th annual international symposium on microarchitecture (MICRO-38),* November 2005, Barcelona Spain.

Heitmeyer, Constance, and Dino Mandrioli. *Formal Methods for Real-Time Computing. Trends in software, 5.* Chichester: John Wiley, 1996.

Hennessy, John, and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 2003.

Hochbaum, Dorit. *Approximation Algorithms for NP-hard problems*. Boston, PWS publishing co. 1996.

Horowitz, Ellis, Sartaj Sahni, and Sanguthevar Rajasekaran. *Computer Algorithms*. Computer Science Press, 1997.

Hubbard, Sarah. *Detection and recognition invention aids target recognition for military pilots*. AFRL News, October 2003.

1394 Trade Association. *1394-based Digital Camera Specification, V. 1.20*. IEEE 1394 Trade Association. July 23, 1998.

Jones, Michael, Daniela Rosu, and Marcel-Catalin Rosu. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities." *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, October 1997.

Judd, Tom. *A Personal Dead-Reckoning Module*. ION GPS, 1997.

Juvva, Kanaka. *Real-Time Systems, Dependable Embedded Systems*. Course 18-849b, Carnegie Mellon University. Spring 1998.

Kaplan, Elliott, and Christopher Hegarty. *Understanding GPS: Principles and Applications (2nd edition)*. Artech House Publishing, Nov. 2005.

Ke, Yan, and Rahul Sukthankar. "PCA-SIFT : A More Distinctive Representation for Local Image Descriptors." *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.

Kelly, Francis, and Anil Kokaram. "General Purpose Graphics Hardware for Accelerating Motion Estimation." *Irish Machine Vision and Image Processing Conference (IMVIP)*, Sept 2003.

Kelly, Francis, and Anil Kokaram. *Fast Image Interpolation for Motion Estimation using Graphics Hardware*. IS&T/SPIE Electronic Imaging - Real-Time Imaging VIII, vol 5297, pp. 184-194. May 2004.

Kilgard, Mark. *The OpenGL Utility Toolkit Programming Interface*. Silicon Graphics, Nov. 13, 1996.

Knaden, Marcus. *No Place Like Home: Ant Navigation Skills Used in Robot Navigation*. Science Daily, April 2006.

Knuth, Donald. *Big Omicron and Big Omega and Big Theta*. SIGACT News, 8(2):18-24, April-June 1976.

Kopetz, Hermann. *Real-Time Systems, Design Principles for Distributed Embedded Applications, Chpt. 10-11*. Klower Academic Publishers, 1997.

Kopetz, Hermann. *The Time-Triggered Architecture*. TU Vienna, Austria, 2000.

Kopetz, Hermann. *Real-time operating systems*. TU Wien, 2002.

Intel Corporation. *Open Source Computer Vision Library*. http://www.intel.com/technology/computing/opencv/index.htm. Last accessed 15 May, 2007.

Ladson, D'Andre. *Boosting Power: Not enough to extend WLAN range*. Texas Instruments technical whitepaper, 2006.

Lakhotia, Arun, Guna Seetharaman, Anthony Maida, Adam Lewis, Suresh Golconda, Amit Puntambekar, and Pablo Mejia. *Technical Overview of CajunBot (2005)*. Submitted to DARPA Grand Challenge, 2005.

Larman, Craig. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development.* Prentice Hall, 2005.

Lindeberg, Tony. *Scale-Space Theory: A Basic Tool for Analyzing Structures at Different Scales.* Journal of Applied Statistics, 1994.

Link, Johannes. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kauffman Publishing, April, 2003.

Lohmann, Kenneth. *Regional Magnetic Fields as Navigational Markers for Sea Turtles*. Science, pp. 364-6. 12 Oct. 2001.

Landing, Don, Mark Munson, Stephan Nadeau, Seth Landsman, Brigit Schroeder, Eddy Cheung, Virgil Zetterlind, and Dan Potter. *Sensor Data Analysis Framework Research Roadmap. Integrating Stream Processing and Persistent Data Retrieval*. Mitre Technical Report 06B0000255, Feb 2006.

Lowe, David. "Object Recognition from Local Scale-Invariant Features." *In Proceedings of the International Conference on Computer Vision, Sept. 1999*. Corfu, Greece.

Lowe, David. *Distinctive Features from Scale-Invariant Keypoints*. International Journal of Computer Vision, January, 2004.

Macedonia, Michael. *The GPU enters computing's mainstream*. Computer. October, 2003.

Mansinghka, Vikash. *SIFT SLAM Vision Details*. MIT 16.412J, 2004.

Marques, Lino, Urbano Nunes, and Aníbal T. de Almeida. *Olfaction-based robot navigation*. Thin Solid Films, Volume 418, Issue 1, pp 55-58. October 2002.

Maybeck, Peter S. *Stochastic Models Estimation and Control, Vol II*. Academic Press, Inc., Orlando, Florida 32887, 1979.

McCool, Michael, and Stafanus DuToit. *Metaprogramming GPUs with Sh*. AK Peters Publishing, July 2004.

Metelitsa, Boris. *Comparing Software Development Approaches for General Purpose GPU Computing*. 21st Computer Science Seminar, 2005.

Misra, Pratap, and Per Enge. *Global Positioning System: Signals, Measurements and Performance (2nd edition).* Ganga-Jamuna Press, 2006.

Moore, Andrew. *An Introductory Tutorial on KD-Trees*. Technical Report 209, Computer Laboratory, University of Cambridge, 1991.

Moore, Gordon. *Cramming more components onto integrated circuits*. Electronics Magazine 19 April 1965.

Mourikis, Anastasios, and Stergios Roumeliotis. *Performance Analysis of Multirobot Cooperative Localization*. IEEE Transactions on Robotics, 22(4), Aug. 2006, pp. 666-681.

Mourikis, Anastasios, and Stergios Roumeliotis. "A Multi-State Constrained Kalman filter for Vision-aided Inertial Navigation." *In Proc. 2007 IEEE International Conference on Robotics and Automation (ICRA'07)*, Rome, Italy, Apr. 10-14, pp. 3565-3572.

Neider, Jackie, Tom Davis, and Mason Woo. *The OpenGL Programming Guide (Red Book)*. Addison-Wesley Publishing Company, 1994.

Nowozin, Sebastian. *Auto-pano SIFT: Making Panoramas Fun*. http://user.cs.tu-berlin.de/~nowozin/autopano-sift/. Last accessed November, 2006.

NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide v 0.8.2*. April 24, 2007.

The OpenThreads Project. *OpenThreads project home page*. http://openthreads.sourceforge.net/. Last accessed 15 May 2007.

Panzieri, Stefano, Federica Pascucci, and Giovanni Ulivi. *An Outdoor Navigation System Using GPS and Inertial Platform*. IEEE/ASME Trans. on Mechatronics, vol. 7, n. 2, pp. 134-142, 2002, IEEE, USA.

Panzieri, Stefano, Federica Pascucci, and Giovanni Ulivi. "Vision based navigation using Kalman approach for SLAM." *11th Int. Conf. on Advanced Robotics*, Coimbra, Portugal, 2003.

Patterson, David, John Hennessy. *Computer Architecture: A quantitative Approach*. Morgan Kaufmann, Second Edition, 1996.

Peakstream Inc. *The Peakstream Platform: High Productivity Software Development for Multi-Core Processors*. March 2007.

Peercy, Mark, Mark Segal, and Derek Gerstmann. "A Performance-Oriented Data Parallel Virtual Machine for GPUs." *SIGGRAPH*, 2006.

Pinto, Livio, Gian Forlani, and Daniele Passoni. *Experimental Tests on the Benefits of a More Rigorous Model in IMU/GPS System Calibration*. National Research Program COFIN, Ministry of the University and Scientific Research of Italy, 2002.

Rost, Randi. *The OpenGL Shading Language*. Addison Wesley Publishing Company, 2006.

Roumeliotis, Stergios, Paolo Pirjanian, and Maja Mataric. "Ant-Inspired Navigation in Unknown Environments." *In Proc. 2000 AAAI International Conference on Autonomous Agents*, Barcelona, Spain, June 3-7, pp. 25-26.

Roumeliotis, Stergios. *Robust Mobile Robot Localization: From Single-Robot Uncertainties to Multi-Robot Interdependencies*. Ph.D. Thesis, Electrical Engineering Department, University of Southern California, CA, May 2000

Roumeliotis, Stergios, Andrew Johnson, and James Montgomery. "Augmenting Inertial Navigation with Image-Based Motion Estimation." *In Proc. IEEE International Conference on Robotics and Automation*, pp. 4326-33, Washington DC, May 11-15, 2002.

Se, Stephen, David Lowe, and Jim Little. *Vision-based Global Localization and Mapping for Mobile Robots.* IEEE Transactions on Robotics, Volume 21, Issue 3, pages 364-375, June 2005.

Se, Stephen, Tim Barfoot, and Piotr Jasiobedzki. "Visual Motion Estimation and Terrain Modeling for Planetary Rovers." *Proceedings of the International Symposium on Artificial Intelligence for Robotics and Automation in Space (iSAIRAS)*, Munich, Germany, September 2005.

Shanwad, U.K., V.C. Patil, G. S. Dasog, C.P. Mansur, and K. C. Shashidhar. "Global Positioning System (GPS) in Precision Agriculture." *In Proceedings of Asian GPS conference*, New Delhi, India. 24-25 October, 2002.

Sharpe, Titus, and Barbara Webb. "Simulated and situated models of chemical trail following in ants." *Proc 5th International Conference Simulation of Adaptive Behavior*, 1998.

Shimpi, Anand. *3DLabs' P10 Visual Processing Unit - When a CPU & GPU Collide*. http://www.anandtech.com/video/showdoc.html?i=1614, May, 2002.

Sim, Robert, Pantelis Elinas, Matt Griffin, and James Little. “Vision-based SLAM using the Rao-Blackwellised Particle Filter.” *Proc of Reasoning with Uncertainty in Robotics*, Edinburg, Scotland, 2005.

Sim, Robert, Matt Griffin, Alex Shyr, and James Little. “Scalable real-time vision-based SLAM for planetary rovers.” *Proceedings of the IEEE IROS Workshop on Robot Vision for Space Applications*, Edmonton, AB, 6 pages, 2005.

Simmons, Jim. *Bat Sonar and Anti-Submarine Warfare*. Science Daily, excerpt from the Office of Naval Research news release, April 2002.

Sinha, Sudipta, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. “GPU-based Video Feature Tracking and Matching.” *EDGE 2006, workshop on Edge Computing Using New Commodity Architectures*, Chapel Hill, May 2006.

Snay, Richard. *GPS Accuracy, Ionosphere effects on horizontal position*. NGS/NOAA Report on removal of GPS Selective Availability, 2000. http://www.ngs.noaa.gov/FGCS/info/sans_SA/iono/. Last accessed May, 2007.

Dahlkamp, Hendrik, Adrian Kaehler, David Stavens, Sebastian Thrun, and Gary Bradski. *Self-Supervised Monocular Road Detection in Desert Terrain*. Technical Report for the 2005 Darpa Grand Challenge. 2006.

Surmann, Hartmut, Andreas Nuchter, and Joachin Hertzberg. *An Autonomous Mobile Robot with a 3D Laser Range Finder for 3D Exploration and Digitalization of Indoor Environments.* Robotics and Autonomous Systems 45, pp 181-198, 2003.

Tarditi, David, Sidd Puri, and Jose Oglesby. *Accelerator: simplified programming of graphics processing units for general-purpose uses via data-parallelism*. Microsoft Research Technical Report MSR-TR-2005-184, December 2005.

Thompson, Chris, Sahngyun Hahn, and Mark Oskin. “Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis.” *In 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35),* 2002.

Thrun, Sebastian, Maren Bennewitz, Wolfram Burgard, Armin Cremers, Frank Dellaert, Dieter Fox, Dirk Hahnel, Charles Rosenberg, Nicholas Roy, Jamieson Schulte, and Dirk Schulz. *Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva*. Journal of Robotics Research, July 2000.

Thrun, Sebastian, Daphne Koller, Zoubin Ghahramani, Hugh Durrant-Whyte, and Andrew Ng. *Simultaneous Mapping and Localization With Sparse Extended Information Filters: Theory and Initial Results*. International Journal of Robotics Research, 23(7-8), 2004.

Thrun, Sebastian, Michael Montemerlo, Daphne Koller, Ben Wegbreit, Juan Nieto, and Eduardo Nebot. *FastSLAM: An Efficient Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association*. Journal of Machine Learning Research, 2004.

Thrun, Sebastian, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, and Pascal Stang. *Stanley: the robot that won the DARPA Grand Challenge*. Journal of Field Robotics 23(9), 661–692, 2006.

Trancoso, Pedro, and Maria Charalambous. "Exploring Graphics Processor Performance for General Purpose Applications." *In Proceedings of the Euromicro Symposium on Digital System Design, Architectures, Methods and Tools (DSD 2005)*, IEEE Computer Society, pp. 306-313, Porto, Portugal, August 2005.

Trawny, Nikolas, Anastasios Mourikis, Stergios Roumeliotis, Andrew Johnson, James Montgomery. *Vision-Aided Inertial Navigation for Pin-Point Landing using Observations of Mapped Landmarks*. Journal of Field Robotics, 24(5), May 2007, pp. 257-378.

Trentacoste, Matthew. *Implementing Performance Libraries on Graphics Hardware*. CMU undergraduate honor's thesis, 2003.

Tripp, Steven. *Cognitive Navigation: Toward a biological basis for instructional design*. Educational Technology & Society 4, 2001.

Keskinpala, Turker, Mitchell Wilkes, Kazuhiko Kawamura, and Bugra Koku. "Knowledge-Sharing Techniques for Egocentric Navigation." *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC),* Washington DC, October 5-8, 2003, pp. 2469-2476, 2003.

Vaughan, Richard T., Kasper Stoy, Gaurav Sukhatme, and Maja Mataric. *LOST: Localization-Space Trails for Robot Teams.* IEEE Transactions on Robotics and Automation, Vol 18 No. 5, 2002.

Veth, Michael. *Stochastic Constraints for Fast Image Correspondence Search with Uncertain Terrain Model.* IEEE Transactions on Aerospace Electronic Systems, 42(3):973-982, July 2006.

Vizard, Frank. *Safeguarding GPS*. Scientific American, April 14 2003.

Wang, Chieh-Chih, Chuck Thorpe, and Sebastian Thrun. "Online Simultaneous Localization and Mapping with Detection and Tracking of Moving Objects: Theory and Results from a Ground Vehicle in Crowded Urban Areas." *IEEE Int. Conf. on Robotics and Automation*, May, 2003.

Wehner, Rudiger, Barbara Michel, and Per Antonsen. *Visual Navigation in Insects: Coupling of egocentric and geocentric information.* Journal of Experimental Biology, 1996.

Welch, Greg, and Gary Bishop. "An Introduction to the Kalman Filter." *SIGGRAPH* 2001, Course 8, 2001.

Wirth, Niklaus. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

Zhang, Dong-Qing, and Shih-Fu Chang. *Detecting Image Near-Duplicate by Stochastic Attribute Relational Graph Matching with Learning*. DVMM Technical Report, Dept of E.E., Columbia University, July 2004.

## Vita

Captain Jordan Fletcher graduated from San Bernardino High School, California in 1997. He attended Tulane University, New Orleans, where he was awarded a Bachelor's of Science in Computer Engineering in 2001. He received his commission from AFROTC Detachment 320 at Tulane University, 19 May, 2001, where he was recognized as a "blue-chip" graduate from the office the Secretary of the Air Force.

His first assignment was as the Assistant Regional Director of Admissions, "Goldbar" recruiter, for the Air Education and Training Command Southwest Region from 2001 to 2002. He attended Basic Communications Officer Training School at Keesler Air Force Base, where he was recognized as a Distinguished Graduate in 2002. Captain Fletcher served as the lead Electrical Engineer and Flight Commander at the 51st Combat Communications Squadron, Robins Air Force Base, Georgia, from 2002 to 2005. While stationed at Robins, he deployed to Ramstein Air Base, Germany and Constanta Air Base, Romania, in support of Operation IRAQI FREEDOM. In August 2005, he entered the Graduate School of Engineering and Management at the Air Force Institute of Technology. Upon graduation, he will be leaving the Air Force to begin work on the GPS modernization project for the MITRE Corporation.

| REPORT DOCUMENTATION PAGE | *Form Approved* *OMB No. 074-0188* |
|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)* 06-14-2007 | **2. REPORT TYPE** Master's Thesis | **3. DATES COVERED** *(From – To)* August 2005 – June 2007 |
| **4. TITLE AND SUBTITLE** REAL-TIME GPS-ALTERNATIVE NAVIGATION USING COMMODITY HARDWARE | | **5a. CONTRACT NUMBER** |
| | | **5b. GRANT NUMBER** |
| | | **5c. PROGRAM ELEMENT NUMBER** |
| **6. AUTHOR(S)** Jordan L. Fletcher, USAF | | **5d. PROJECT NUMBER** |
| | | **5e. TASK NUMBER** |
| | | **5f. WORK UNIT NUMBER** |
| **7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)** Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENG) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865 | | **8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCS/ENG/07-02 |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** AFRL Munitions Directorate 101 W. Eglin Blvd. Eglin AFB, FL 32542-6810 | | **10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/MN |
| | | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**13. SUPPLEMENTARY NOTES**
**Sponsor Name: Dr. Timothy J. Klausutis, (comm.) 850-883-0887**

**14. ABSTRACT**

Modern navigation systems can use the Global Positioning System (GPS) to accurately determine position with precision in some cases bordering on millimeters. Unfortunately, GPS technology is susceptible to jamming, interception, and unavailability indoors or underground. There are several navigation techniques that can be used to navigate during times of GPS unavailability, but there are very few that result in GPS-level precision. One method of achieving high precision navigation without GPS is to fuse data obtained from multiple sensors.
This thesis explores the fusion of imaging and inertial sensors and implements them in a real-time system that mimics human navigation. In addition, programmable graphics processing unit technology is leveraged to perform stream-based image processing using a computer's video card. The resulting system can perform complex mathematical computations in a fraction of the time those same operations would take on a CPU-based platform. The resulting system is an adaptable, portable, inexpensive and self-contained software and hardware platform, which paves the way for advances in autonomous navigation, mobile cartography, and artificial intelligence.

**15. SUBJECT TERMS**

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** Michael Veth, Maj, USAF |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | **19b. TELEPHONE NUMBER** *(Include area code)* |
| U | U | U | UU | 130 | (937) 255-6565, ext 4541 (Michael.veth@afit.edu) |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std. Z39-18